DTIC FILE COPY

(4)

AD-A203 278

CLOCS Architecture Reference Documents

TR88-021

May 1988

Contract N00014-86-K-0680

Mark C. Davis
Bill O. Gallmeister

DTIC
ELECTE
JAN 2 6 1989
S D
D

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

The University of North Carolina at Chapel Hill
Department of Computer Science
CB#3175, Sitterson Hall
Chapel Hill, NC 27599-3175

Copyright ©1988 Mark C. Davis and Bill O. Gallmeister

88 11 18 176

CLOCS Architecture Reference Documents

Mark C. Davis
Bill O. Gallmeister

CB # 3175, Department of Computer Science [1]
University of North Carolina
Chapel Hill, NC 27599-3175

DTIC
COPY
INSPECTED
1

May 6, 1988

Accesion For
NTIS CRA&I ✓
DTIC TAB
Unannounced
Justification
By per ltr
Distribution /
Availability Codes
Dist | Avail and/or Special
A-1

[1] This Research was supported by Office of Naval Research Contract N00014-86-K-0680.

Contents

Introduction **1**

1 Overview **2**

1.1 Highlights 2

1.2 Instruction Formats 3

1.3 Programming Model 5

1.4 Data Formats 7

1.5 Operations 8

1.6 Implement Notes 11

2 Memory Management Unit **12**

2.1 Organization of the Memory Management Unit 12

2.2 Contents of the MMU Word 13

2.3 MMU Operations 16

2.4 MMU Exceptions 18

2.5 Implementing Common Virtual Memory Operations 19

2.6 The MMU Designs We Discarded 19

3 Assembler Language **22**

3.1 Memory Architecture 22

3.2 Data Types 23

3.3 Instruction Syntax 23

3.4 Instruction List 24

3.5 Addressing Modes 24

3.6 Conditional Skip 26

3.7 Instruction List 28

4 Compiler Description **34**

4.1 Class Project 34

4.2 Fast Context Switch 34

4.3 CLOCS Architecture 35

Introduction

This technical report combines several CLOCS architecture papers from the academic year 1987-88. It contains an overview, a detailed discussion of the memory management unit, a description of an assembler language and specifications for compiler writers. Because of the nature of each of the four chapters, some areas are repeated in several chapters.

Chapter 1

Overview

The CLOCS architecture is an attempt to remove the highest layers of memory hierarchy to reduce the effort of switching execution from one task to another. As a result, the CLOCS architecture is a very simple one, with only memory to memory instructions.

1.1 Highlights

The major design consideration of this machine is to switch context in as short a time as possible. All instructions are memory to memory. The cpu implements only one data type, a 64 bit two's complement fixed point number. Instructions are one 64 bit word. The program context includes a status register containing a program counter, a process identification number and various flags.

1.1.1 Noteworthy

The machine can switch context in less than one processor cycle. The overhead to switch from one process to another is approximately equal to one third processor cycle. This feature results in much higher performance when many context switches are occurring and satisfies the requirements of many applications requiring very fast context switching.

1.1.2 Peculiarities

The only state inside the central processing unit(CPU) is a program status word. All data operations are memory to memory. One result of this is that all parameter passing must be through memory.

All memory accesses are 64 bits long. All instructions are 64 bits long and the only

5	5	6	24	24
OP	AM	Flags	Operand 1	Operand 2

OP Operation Code

AM Address Mode for Operand 1 and Operand 2

Figure 1.1: CLOCS Instruction Format

numerical data type is 64 bits long.

1.1.3 History

This architecture was conceived by Mark Davis in response to the challenge "if I where to design RISC, this is how I would do it." The architecture was designed by Mark Davis and Bill Gallmeister.

1.2 Instruction Formats

There is one instruction format, a one word format. Figure 1 shows the instruction format.

1.2.1 Structure

Each instruction contains an operation code, a standard set of addressing mode flags, a operation dependent flag field and specification for two operands.

1.2.2 Address specification

The 24 bit operand in the instruction may be used as immediate data or may be combined with a 16 bit segment identifier (SID) to form a 40 bit virtual address. Each processes has a primary instruction SID (PSID) and a primary operand SID (OSID) assigned. Indirect addressing is also supported and during indirect addressing, the OSID or ISID may be overridden to access any word in the teraword (2^{40}) address space.

1.2.3 Formats

The instruction consists of

- (5 bits) operation code
- (5 bits) addressing flags, described in sect 3.4.
- (6 bits) operation flags, described in section 3.3.2 to 3.3.5 below
- (24 bits) operand 1 specification
- (24 bits) operand 2 specification

Branch and Trap Operations

For the trap operation, the flags are used to determine whether the trap will occur by evaluation of operand 2. Addressing mode for operand 1 is not used. For the Branch operation, the flags determine whether the branch will be taken by evaluation of operand 2.

Arithmetic Operations

In arithmetic operations, the flags determine whether to skip the next instruction by evaluation of the result of the operation.

Shift Operations

For shift operations, the flags constitute the number of bits to shift operand 1 before storing it in operand 2.

Logical Operations

During logical operations, the flags determine whether to skip the next operation and whether to perform the operation on the entire word or on just one byte.

1.2.4 Spaces and Addressing

The only numbered address space is to main memory. The cpu deals with virtual address and these are converted to physical addresses by the Memory Management Unit (MMU). There are 40 bits in a virtual address for an address space of 1 teraword. There are 30 bits in a physical address for a space of 1 gigaword.

1.3 Programming Model

The programming model is pretty simple. All operations are memory to memory. Any special devices added should be memory mapped. The program counter is even memory mapped.

1.3.1 Working storage

There is no working storage. Everything is kept in main memory.

1.3.2 Memory Name Space

The memory name space is a linear sequence of one word (64 bit) entries. The instructions address the memory using virtual addresses that are 40 bits long. These virtual addresses are changed into a 30 bit physical address by the MMU. The formation of these addresses is discussed at the end of the next section.

1.3.3 Addressing Modes and Address Calculation

CLOCS supports seven addressing modes, all of which are available for use with operand one, and four of which are available for operand two. All modes used for operand 2 have a high order bit of 0, so only the 2 lowest order bits appear in the instruction. In these descriptions, "+" means catenate the two values. Before describing each of the addressing modes, address formation will be discussed.

Virtual Address Formation

A virtual address is forty bits long, and it may formed in two ways. A 16 bit segment identifier (SID) and a 24 bit offset may be combined to form the address. Each process has a default segment assigned for both instructions and data. The MMU stores these segment identifiers and uses the process identifier to find the correct segment identifier. The 24 bit offsets appear in the instructions or may be obtained from main memory. A second method of providing the 40 bit virtual address is to get it from the 40 low order bits of a memory location.

Physical Address Formation

The MMU can calculate a physical address in one of two ways. In the first case, the CPU provides a process identification number and a 24 bit offset. The MMU associatively looks up the physical page corresponding to the default segment for the given process and the 12

high order bits of the offset. In the second case, the CPU provides the entire 40 bit address. Then MMU then associatively looks up the physical page corresponding to the 28 high order bits (16 SID + 12 from offset). After the physical page has been identified by either method, the MMU verifys that the requested operation (read or write) is authorized for this process. If it is, the 30 bit physical address is formed from 18 bits of the physical page and the 12 low order bits from the virtual address.

Opnd

(Opnd1, Opnd2)

Operand := FETCH (OSID + Opnd)

OSID, the operand SID, is catenated to the high-order end of Opnd to provide a full 40-bit virtual operand address from which the operand is fetched. This is CLOCS' "direct mode" of addressing.

@Opnd

(Opnd1, Opnd2)

Operand := FETCH (OSID + FETCH(OSID + Opnd))

OSID is catenated to Opnd to form an virtual address. From this address is fetched a 24-bit offset. This offset is catenated with OSID to form the virtual operand address. This is CLOCS' "indirect mode" addressing.

%Opnd

(Opnd1, Opnd2)

Operand := FETCH (0 Segment + Opnd)

The operand is catenated to a SID of zero to arrive at the virtual operand address. This provides rapid zero-page addressing, but otherwise is identical to Direct Addressing.

%@Opnd

(Opnd1, Opnd2)

Operand := FETCH(FETCH (OSID + Opnd))

OSID is catenated to Opnd to form an virtual address; from this address a word con-

taining a 40-bit address is fetched to form a virtual address into any page. This is indirect addressing FROM the process' page, INTO any page.

%@%Opnd

(Opnd1 ONLY)

Operand := FETCH(FETCH (0 Segment + Opnd))

Opnd is catenated with the zero page SID to form a virtual address; from this address in the zero page a word containing a 40-bit address is fetched. This virtual address is used to fetch data in any page. This is indirect addressing FROM the zero page, INTO any page.

@%Opnd

(Opnd1 ONLY)

Operand := FETCH(OSID + FETCH (0 Segment + Opnd))

Opnd is catenated with the zero page SID to form an virtual address. From that address, a 24-bit offset is fetched. This offset is catenated with the OSID to form the virtual operand address. This is indirect addressing FROM the zero page, INTO the process' page. (We do not see a great need for this instruction, however we put it in for symmetry. The compiler (and the compiler writers) can tell us if it is useful.)

<Opnd

(Opnd1 ONLY)

Operand := Opnd

Opnd is a 24-bit immediate operand.

1.4 Data Formats

We claim to have only one data format, but actually the architecture supports two formats: 64 bit fixed point and 64 bit floating point.

1.4.1 Fixed Point

The normal data format is a two's complement 64 bit fixed point number.

1.4.2 Floating Point

The floating point data format is IEEE 754 64 bit format.

1.4.3 Character

A word may also be considered as an array of 8 characters. The logical operations have the ability to address each byte separately.

1.5 Operations

The CLOCS architecture has 18 operations defined. There are 5 fixed arithmetic, 4 floating point arithmetic, 6 logical, 1 sequencing and 2 supervisory.

1.5.1 Decision

CLOCS has no specific decision operations. Instead, a conditional branch is provided and all arithmetic and boolean logical operations incorporate conditional skip. The behavior of this sequencing will be discussed with each other category of operation.

1.5.2 Data Operations

Data operations are partitioned into fixed and floating point arithmetic, boolean logic operations, and shifts.

Arithmetic Operations

CLOCS supports 64 bit fixed point arithmetic. For operations resulting in more that 64 bits, such as multiply, the high order bits are lost. Similarly, the fractional result of a divide is lost. Indication of multiply overflow is available to the programmer. The program may use the remainder instruction to detect and manipulate fractional divide results.

Operation codes have been set aside in CLOCS for floating point arithmetic. We plan that early implementations of CLOCS would not include floating point hardware, and these

instructions would cause "unknown operation" faults, so the operating system could then perform the floating point operations.

For both types of arithmetic, the following instruction is skipped if any of six conditions are flagged in the instruction and are true. These conditions are:

LT result of operation less than zero.

GT result of operation greater than zero.

EQ result of operation equal to zero.

NO result of operation did NOT overflow.

NU result of operation did NOT underflow.

NZ result of operation was NOT a divide-by-zero.

Note that these conditions (or their negation) cannot be true for some operations. For instance, it is not possible to get underflow unless a floating point operation is being performed.

Boolean Logic Operations

CLOCS provides AND, OR and XOR logical operations. These operations may apply to an entire word or to one 8 bit byte within that word.

Skips for boolean logical operations occur for two possible conditions:

EQ result of operation equal to zero.

NZ result of operation is not zero.

Shifts

CLOCS provides shift left, shift right, and shift right arithmetic (extends two's complement sign). The number of bits to be shifted (from 0 to 63) is specified in the instruction. Please note that a zero bit shift may be used as a move. The shift instructions have no conditional skip.

1.5.3 Sequencing

The sequence of instructions is controlled by the branch instruction, supervisor calls and interrupts.

Branches

The branch instruction is conditional, based on the contents of the second operand (evaluated either as a fixed or floating point number):

Interrupt and Supervisor Call

CLOCS has a large number of interrupt vectors. On a Supervisor Call (Trap) or an interrupt, the old status word is saved and the new status word for that supervisor call or interrupt is loaded. Interrupts are grouped into maskable levels, and presumably, each interrupt status word would mask that level of interrupt long enough to move the save status word out of the way (to make interrupts reentrant).

The Supervisor call instruction has a conditional execution. If a flag is set and the corresponding condition is true, then execution continues at the address specified in the instruction. Otherwise, the following instruction is executed.

LT result of operation less than zero.

GT result of operation greater than zero.

EQ result of operation equal to zero.

1.5.4 Supervisory

Two supervisory instructions are provided. The trap instruction conditionally causes the execution of a supervisor call at a trap vector location. This qualifies as a supervisory instruction because the status word is directly loaded from the trap vector, allowing the machine to change to operating system process identification number. Condition flags for this instruction are the same as for the branch.

The load operand segment instruction allows a program to use a different default data segment. If the identified segment is not available to the process, the cpu will cause a fault.

Although, not specifically allocated as a supervisory instruction, moving data to the certain addresses from ffff.ff0000 to ffff.ffffff causes changes to the cpu. For example, writing to ffff.ffffff changes the status word. That memory location is owned by the operating system process and cannot be written by any other process.

1.5.5 Input and Output

Input and output devices are memory mapped, so no special operations are provided to manage them. The memory mapping is down in the memory address range ffff.ff0000 to ffff.ffefff. A special set of addresses is provided so virtual memory and cache algorithms will not interfere with proper device operation.

1.6 Implement Notes

The architecture leaves two major hooks to permit single chip implementations with a reasonable numbers of transistors. First, operations are defined for floating point, but no hardware support is required. Under normal circumstances, floating point will be emulated by operating system routines. The second hook concerns the size of the mmu. Although provision has been made for a very large number of mmu registers, a machine could be built with very few registers, perhaps with only four registers. Although scrimping on the mmu will save chip area, it will have a major impact on context switching performance; therefore, we recommend having at least one mmu register for each page of physical memory installed in the machine.

Hopefully, the implementations of CLOCS will be heavily pipelined. A 4 stage pipeline with interlocks or about a 7 stage pipeline without interlocks seems reasonable. Note, that pipelining will increase context switch latency, which may be significant if a realtime task has to be serviced in less than 20 cycles (it is not clear how you can write a scheduler for that, but it is a consideration). Also, caching inside the cpu may effectively improve performance. Caching intermediate results to avoid memory references and short circuiting pipeline latency may both be major average performance improvers.

Chapter 2

Memory Management Unit

2.1 Organization of the Memory Management Unit

The CLOCS Memory Management Unit(MMU) must support virtual memory with as many contexts as possible. We used this guiding principle: "If information for a process is in main memory, it must be accessed with no context switch penalty." Another design requirement was that the MMU support lightweight processes, because an important application, real time operating systems, frequently use lightweight processes [2]. This meant that the MMU would provide a sharable address space with protection for the space owned by a each process. Provisions for protected sharing of memory between two processes was also an important requirement for real time.

2.1.1 What the MMU Does

The purpose of the MMU is to support virtual memory for the CLOCS computer system. It does that by taking an address specification from the cpu, determining the corresponding physical address, checking that the current process has permission for the requested memory operation, and maintaining information of use to the operating system. The real work of address translation is done to the physical page level; the low order 12 bits of the virtual address are used as the low order bits of the physical address. Permission is granted for three possible categories of operations to be applied to three types of pages: read only, read or write, and execute only. The MMU also keeps records of access and writing to physical pages. It records when a physical page has been read or written, USED, so the operating system can later determine the best page to swap out using the a common algorithm for virtual memory. The MMU also records when a "read or write" type physical page has been changed, DIRTY, so the operating system can avoid unnecessary saving of pages to backing store.

2.1.2 External Appearance of the MMU

All of the information to determine physical addresses, check permission, and remember physical memory status is kept in 64 bit registers in the MMU. These MMU registers are memory mapped at beginning at location ffff.f00000, and are protected from user processes. Only the superuser, PID = 0, may change them. Since each of the registers contains information about one physical page, the MMU should contain at least as many registers as the computer system has physical pages. In order to meet the design guiding principle concerning memory access time, an excess of MMU registers should be provided for shared pages. Memory address ffff.feffff is reserved for the number of MMU registers installed. The MMU intercepts references to this location and provides the number. This same memory location is also used as a command register. CLOCS can address up to 262,144 pages (2^{18}), but since this corresponds to 1,073,741,824 words (8 gigabytes) of memory, most machines will have less physical memory and need much fewer than 263,144 MMU registers. In the absence of data, we estimate that an additional 10% of MMU registers over the maximum number of expected physical pages will be adequate.

2.1.3 Physical Page Status

Part of each MMU register are some bits to indicate the status of the referenced virtual and physical page. The use status and written or DIRTY status is maintained for the physical page. More than one MMU register may refer to a physical page; this is the way that memory would be shared. The MMU must provide the correct status for a physical page when an MMU register is read. For example, MMU register ffff.f00001 and ffff.f00009 both point to physical page 4. A write is made using the entry at ffff.f00009. If the register at ffff.f00001 is subsequently read, its status will indicate that the page is DIRTY even though no write was made using that MMU entry.

Implementor may accomplish this magical updating of physical page status in any manner, but one solution is suggested. An auxiliary memory with a two bit word for each physical page stores the correct status of each physical page. During routine memory operations, the status of a page would be updated in parallel with the memory operation. When an MMU register is read, the physical page address in the MMU register is used to access the auxiliary memory. The use and DIRTY bits from the auxiliary memory are used to update the MMU register before it is provided to the CPU. As long as the page status could be fetched and the MMU register status updated in the time of a main memory fetch, the organization would not effect performance.

2.2 Contents of the MMU Word

The MMU registers are divided into six fields. Before we examine the MMU registers, a quick review of terms. Each abbreviation is followed by the number of bits.

PID (14) Process identifier.

SID (16) Segment identifier.

OSID (16) the default operand SID.

ISID (16) the default instruction SID.

VO (24) Virtual Offset.

PC (24) Program Counter, a VO.

OPND (24) Operand of in an instruction, a VO.

VP (12) Virtual page.

PP (18) Physical page.

PO (12) Physical Offset, the low order bits of VO.

VA (40) Virtual address, SID+VO.

PA (30) Physical address, PP+PO.

Each MMU register (or entry) is a 64 bit word. The MMU may store the information for each entry in any convenient format, but it must appear as a 64 bit memory address to the cpu with the following format:

PID 14 bits - Process Identification Number

Flags 4 bits - Permissions and Physical Page State

SID 16 bits - Segment Identifier

VP 12 bits - Virtual Page

PP 18 bits - Physical Page

2.2.1 Field Sizing Considerations

The sizing of fields followed from the portions of the architecture which was defined before MMU design was completed. The SID was set at 16 bits. We wanted at least 1 gigaword of physical storage, so the physical address required 30 bits. Flags required about 4 bits. We wanted to have 16 bits for PID and physical pages of 1024 words. Since the operand address size was 24 bits, this physical page size would have required the VP to be 14 bits and the PP to be 20 bits. The 34 bits for VP and PP plus the 16 bits for SID leaves only 14 bits for flags and PID. A 10 bit PID, allowing only 1024 active processes was deemed too restrictive, so we settled on a 4096 word physical page. This final page size required 12 bits, reducing VP to 12 bits and PP to 18 bits. With this design, the combination of VP, PP, SID requires only 46 bits, leaving 18 bits to be divided between 4 bits of FLAGS and a 14 bit PID. This compromise raised the importance of maintaining the FLAGS field no larger than 4 bits, so the assignments of the flag field bits is discussed below.

2.2.2 Mapping of Word use to Flags

The MMU has to maintain much permission information and status for each physical page. A process may have data access or read permission to a page. A process may have write permission to a page. The page may contain code executable by the specified process. The SID may be the primary SID for the PID. The page may be DIRTY, that is it is a writable page and has been changed since it was paged in. The page may have been accessed since accessed information was updated. If each of these categories of information were to be represented by one bit, the flag field would require 6 bits instead of the allotted 4.

For the discussion of how we saved the two needed bits, I will use the following abbreviations:

R The page is readable by the process

W The page may be written by the process

X The page may be executed by the process

P The SID is the primary SID for this type of page for this process

U This page has been USED

D This page has been written, DIRTY

Many of the combinations do not make sense. To see these nonsensical combinations, we constructed a truth table. A bullet(•) in this table indicates that this is not a viable alternative. A number indicates that this combination of attributes is useful and should be represented in the MMU registers.

									Reason for Elimination
		U	D	D U	P	P U	P D	P D U	
	1	•	•	•	•	•	•	•	Unallocated can't be P,D,U
X	2	3	•	•	4	5	•	•	Executable can't be DIRTY
W	•	•	•	•	•	•	•	•	No Write only pages
W X	•	•	•	•	•	•	•	•	No X and R or W
R	6	7	•	•	•	•	•	•	No D, P without W
R W	8	9	10	11	12	13	14	15	
R X	•	•	•	•	•	•	•	•	No X and R or W
R W X	•	•	•	•	•	•	•	•	No X and R or W

With only 15 usable states to represent, only 4 bits of state will be required. We reorganized the states as shown below. The numbers at the right of the table are the 2 high order bits of the flag field in the MMU word. The numbers at the bottom of the table are the low order bits of the flag field. The numbers inside the table correspond to numbers in the first table.

		USED	Primary	Primary USED	
Executable	2	3	4	5	00
Read Only	6	7	1		01
Read and Write	8	9	12	13	10
Read and Write Dirty	10	11	14	15	11
	00	01	10	11	

With this bit assignment, the third bit becomes the USED bit, the fourth bit is the Primary bit, the first and second bit must be taken together to interpret the permissions. The combination 0110 represents an unassigned physical page.

2.3 MMU Operations

The MMU must perform several operations.

2.3.1 Normal Read and Write

The MMU registers can be read and written by the superuser process, PID = 0. The MMU registers are addressed as normal memory, so the MMU must reconize addresses starting at ffff.f00000 and respond to them rather than trying to calculate a physical address.

Possible exceptions:

- Memory not present
  addressing MMU register not installed
- Memory Permissions Incorrect
  PID $\neq$ 0
- Flag 1101 not permitted
  Unassigned Flag combination

2.3.2 From PID,VP get PP and Check Permissions

When presented with a PID, a VP, and a signal that this fetch is for an operand, the MMU must determine the correct PP and check permissions.

Possible exceptions:

- PID, SID, VP not in MMU
- Memory Permissions Incorrect

2.3.3 From PID,VP get PP and Check Permissions

When presented with a PID, a VP, and a signal that this fetch is for an instruction, the MMU must determine the correct PP and check permissions.

Possible exceptions:

- PID, SID, VP not in MMU
- Memory Permissions Incorrect

2.3.4 From OSID,VP get PP and Check Permissions

When presented with a PID, an SID, a VP and a signal that this is an operand fetch, the MMU must determine the correct PP and check permissions.

Possible exceptions:

- PID, SID, VP not in MMU
- Memory Permissions Incorrect

2.3.5 From ISID,VP get PP and Check Permissions

When presented with a PID, an SID, a VP and a signal that this is an instruction fetch, the MMU must determine the correct PP and check permissions.

Possible exceptions:

- PID, SID, VP not in MMU
- Memory Permissions Incorrect

2.3.6 Change Primary OSID

When directed by the cpu, change the primary OSID to the SID provided on the low order 16 bits on the data bus. This update requires setting the Primary flag on all entries with the PID and new OSID and resetting the Primary flag in all MMU registers with the PID and the old OSID.

Possible exceptions:

- PID, SID not in MMU
  An authorized page is has been paged out
  Thi PID is not authorized to share this page
- Memory Permissions Incorrect
  The new segment identified by SID is not writable

2.3.7 Change Primary ISID

When an instruction fetch is made and the instruction is located in a segment different from the current process' primary ISID, the MMU must store the new SID. When the cpu signals "last branch taken," the MMU must update the stored SID to be the new primary ISID for this process. This update requires setting the Primary flag on all entries with the PID and new SID and resetting the Primary flag in all MMU registers with the PID and the old ISID. The cpu must signal "branch not taken" if a conditional branch is to taken. The MMU may stall if more than one instruction fetch specifies a new ISID before it receives a "last branch taken" signal.

Possible exceptions:

- None

The exception PID, SID, VP not in MMU can not occur for this operation because the MMU must first fetch the new instruction using one of the above operations. If there is an interrupt, the branch instruction will be restarted, so we will always know that the physical page is available. Additionally, the instruction fetch operation will verify that this page contains executable code, so no Memory Permissions Incorrect exception may occur.

2.3.8 Reset USED for All Physical Pages

When the operating system selects a page to swap out of main memory it may use the USED bit. Frequently, the operating system will want all USED bits set to zero. To set the USED bit for all physical pages to zero, write a word with the low order bit of 1 to the memory location ffff.feffff. That location when read contains the number of MMU registers installed.

Possible exceptions:

- None

2.4 MMU Exceptions

Exceptions have been described after each operation.

2.5 Implementing Common Virtual Memory Operations

In this section, I will described how to implement some common virtual memory operations using the primitives provided by the CLOCS MMU.

2.5.1 Write Back Virtual Memory

Before a page may be removed from physical memory, the DIRTY status should be check for any MMU register referring to that physical page. Saving the page on disk before reusing the page is only required when the DIRTY status is set. This method significantly reduces memory traffic because much data memory is read, but not changed before it is paged out.

2.5.2 Copy on Write

Copy on write is an algorithm frequently used by UNIX operating systems and VAX computers. A process is assigned a block of memory containing information or code. As long as it does not change this memory, it shares the memory with another process. As soon as the process attempts to change the memory, the operating system must intervene to make a separate copy for this process, and then allow the change to happen. This facility is very useful for the vfork system call in UNIX. Copy on write may be simulated by assigning the page as a shared, read only page. Shared simply means that the page has more than one MMU register pointing to it. When the process tries to write to the "copy on write" page, the MMU causes an exception. The operating system exception handler then copies the page to an unused physical page. It then corrects the MMU register to point to the new physical page and restarts the user process with the instruction that caused the fault.

2.5.3 Not-Used-Recently Page Replacement

One popular page replacement algorithm is Not-Used-Recently. This technique is described in detail in Deitel [1]. Deitel points out that a USED bit and a DIRTY bit must be maintained for each page, and this information is available from the CLOCS MMU.

2.6 The MMU Designs We Discarded

During MMU design, we considered several schemes: the one described above and anothers. Some of the alternate design were interesting to us or involved important design decisions, so the ones we threw away are described here.

2.6.1 The Second Design - Virtual and Physical Tables

The second design differed in that the MMU contained two tables instead of one. One table, Table1, contained PID, FLAGS, and SID. The other table, Table2, held a Dirty bit, SID and VP. The second table had one entry for each physical memory page, so the PP did not have to be included in the table. The advantage of the second scheme was that it was more proper for support of lightweight processes. The PID, SID, FLAGS relationship was unique. The primary scheme was better in that it could support heavyweight as well as lightweight processes and also could resolve the permissions down to the physical page level. With that scheme, one segment could hold both code and data space on separate pages, so small processes need not take up two segments of address space. The other difference between the schemes was the simplicity of the data structure and duplication of PID's for the primary scheme and duplication of SID's in the secondary scheme.

The final decision of which scheme to use was based on the projected silicon area of the two schemes. We assumed field sizes the same for the two schemes except the secondary scheme needed one extra DIRTY bit. PID, FLAGS, SID and Virtual Page were all associative. This distinction was made because associative bits would require at least 25 % more silicon area to implement. Most associative bit implementations would require about 50 % more area than a nonassociative bit.

To compare the two schemes, we specified a computer system with 4000 pages of physical memory and capable of running 1000 processes. This machine is a typical system to utilize the power of the CLOCS architecture and support large applications. For a machine of this size, the primary scheme required 4500 table entries (one for each physical page plus 500 for memory sharing). Each entry was 64 bits long, 44 of which were associative. The secondary scheme required Table1 with 2500 entries, two for each processes (one data, one code) and 500 extra for memory sharing. Each entry in this table was 34 bits long and all were associative. The second table, Table2, contained 4000 entries, one for each physical page. Each entry was 29 bits long and 28 of them where associative. The table below shows the bits and relative area for the two schemes. The column labeled "Total Relative Area" is the total area of the table in nonassociative bits, assuming that associate bits are 50% larger than nonassociative ones.

MMU Scheme and Table	Associative Bits	Total Bits	Total Relative Area
Secondary Table1	80,000	85,000	125,000
Secondary Table2	112,000	116,000	172,000
Secondary Total	192,000	201,000	297,000
Primary Total	198,000	288,000	387,000

The small additional cost of associative bits and the increased function of the primary scheme, particularly since the primary scheme supported heavyweight processes, a concept used by many available operating systems, settled the decision in favor of the primary scheme.

2.6.2 The Third Design - Some Registers Permanently Mapped

The third MMU design attempted to reduce the number of bits of memory in the MMU and to make some operating system task more efficient by permanently assigning some of the MMU registers to physical pages. In this scheme, memory locations ffff.f00000 to ffff.f3ffff were assigned to physical pages 0 to 262,144, respectively. These memory locations always returned the corresponding physical page number when read, and the physical page was ignored during writes to these MMU registers. The memory from ffff.f40000 to ffff.fefffe could be assigned to any physical page.

The advantages of this third scheme were fewer memory bits in the MMU and a possible improvement in operating system speed. If a computer system had N physical pages and allow for and addition M pages to be shared, then N+M MMU registers would be required. We estimate The third scheme would then save N*18 bit s of memory over the primary scheme. Another advantage for this scheme was improved performance during a naive search for a page to swap out. With the third scheme, a search for a potentially shared page would only require O(M) while the primary scheme would take O(N+M). As estimated above, M would only be 10% of N, so this new scheme would yield an order of magnitude performance improvement. This advantage disappeared, though, when a O(log M) software algorithm was suggested. The data structures and algorithm to attain this superior level of performance are well understood.

With one major advantage of this scheme eliminated, the disadvantages became more persuasive. This scheme of two classes of MMU registers lacks propriety. Although the same operations may be performed on the two types of MMU registers, different actions result. If the systems programmer makes an error, and tries to set the physical page number of one of the permanently assigned MMU registers, the action is ignored and the programmer receives no warning of his error. An additional disadvantage of the third scheme is that the number of shared pages is limited to M. With the primary scheme, all MMU's registers may be used for shared pages, with only the disadvantage that some physical pages may not be accessible, a much more graceful degradation of performance.

Since the only advantage to this scheme was the saving of some memory in the MMU and it introduced such serious impropriety, we selected the primary scheme over it.

Chapter 3

Assembler Language

3.1 Memory Architecture

The CLOCS memory space is all mapped in to one address space. The working store (program counter), Memory Management Unit (MMU) registers and all Input Output devices share the address space with main memory. Refer to the CLOCS Compiler and Assembly Language Description for a more detailed treatment of the architecture.

3.1.1 Memory Scheme

A quick review of terms pertinent to the memory:

PID (14) Process identifier.

SID (16) Segment identifier.

OSID (16) the default operand SID.

ISID (16) the default instruction SID.

VO (24) Virtual Offset.

PC (24) Program Counter, a VO.

OPND (24) Operand of in an instruction, a VO.

VP (12) Virtual page.

PP (18) Physical page.

PO (12) Physical Offset, the low order bits of VO.

VA (40) Virtual address, SID+VO.

PA (30) Physical address, PP+PO.

The MMU translates the combination of PID, SID, VP to PP and checks the PID's permission on that SID+VP combination. The 30 bit Physical Address gives that machine a real memory capability of 1,073,741,824 words (or 8 gigabytes). All accesses to memory are by 64-bit word access only.

3.1.2 Memory-Mapped Access

CLOCS reduces the variety of its instructions by mapping all state information of the machine into the memory space of the processor. Thus, the State Word, consisting of the PC, PID, and Flags, may be found at location ffff.fffff (This is segment ffff, address fffff). The MMU registers begin at ffff.f00000. Location ffff.feffff contains the number of MMU registers installed on this CPU. Input-output devices are mapped into the memory from ffff.ff0000 to ffff.ffefff. The trap and interrupt vectors, likewise, can be found in the this segment, at addresses ffff.fff000 to ffff.ffffe.

3.2 Data Types

CLOCS supports a single arithmetic data type: the 64-bit signed integer represented as a 2's complement. There is provision for an optional data type, a 64 bit IEEE 754 floating point number.

3.3 Instruction Syntax

The input to the assembler is an ascii text file. Each line in the text file contains (1) a machine instruction (2) a assembler directive (3) a label or (4) a comment.

3.3.1 Machine Instructions

A machine instruction consists of zero or more spaces, an operation code abbreviation followed by one or more spaces followed by the operands of the instructions. Operands are separated by commas and must not contain spaces. Operands may be decimal integers, hexadecimal numbers indicated by "0X" as the first two characters, or a label which is a word starting with A-Z, "#", or underscore and containing those characters or digits 0-9. Anything on the line after the operands is considered to be a comment and is ignored.

```
Sub   123,loc22  this is a comment
```

3.3.2 Assembler Directives

Assembler directives instruct the assembler for actions that do not result in generation of an executable machine instruction. Some of the assembler directives are followed by a single operand. The assembler directives are as follows:

.sect section command, begins .text, .data, .rom, or .bss sections

.data2 data command, reserves 1 word of storage. May be followed by a decimal integer or by a character string.

.ext specifies the operand is an external label.

3.4 Instruction List

In this instruction format, the 5 bits of operation code are followed by 5 bits of flags which determine addressing modes for the two operands. The next 6 bits specify flags or a count. The operand, 24 bits long, follows. A number of addressing modes, as described elsewhere in this document, can be applied to the operand(s) by the judicious setting of the addressing mode flags.

3.5 Addressing Modes

CLOCS supports seven addressing modes, all of which are available for use with operand one, and four of which are available for operand two. In each subsection below, the title of the addressing mode appears as the header. After each addressing mode identification is the bit pattern appearing in the instruction to identify that mode. All modes used for operand 2 have a high order bit of 0, so only the 2 lowest order bits appear in the instruction. In these descriptions, "+" means catenate the two values. Next is listed the operands for which it may be used. An example is given of the operand. In these examples, 123 refers to location 123 decimal, and loc22 is a label associated with some storage definition statement in the program. A formal and textual definition of the operand location ends each section.

3.5.1 Opnd - 000

(Opnd1, Opnd2)

```
Sub   123,loc22
```

Operand := FETCH (OSID + Opnd)

OSID, the operand SID, is catenated to the high-order end of Opnd to provide a full 40-bit virtual operand address from which the operand is fetched. This is CLOCS' "direct mode" of addressing.

3.5.2 @Opnd - 001

(Opnd1, Opnd2)

```
Sub    @123,@loc22
```

Operand := FETCH (OSID + FETCH(OSID + Opnd))

OSID is catenated to Opnd to form an virtual address. From this address is fetched a 24-bit offset. This offset is catenated with OSID to form the virtual operand address. This is CLOCS' "indirect mode" addressing.

3.5.3 %Opnd - 010

(Opnd1, Opnd2)

```
Sub    %123,%loc22
```

Operand := FETCH (0 Segment + Opnd)

The operand is catenated to a SID of zero to arrive at the virtual operand address. This provides rapid zero-page addressing, but otherwise is identical to Direct Addressing.

3.5.4 %@Opnd - 011

(Opnd1, Opnd2)

```
Sub    %@123,%@loc22
```

Operand := FETCH(FETCH (OSID + Opnd))

OSID is catenated to Opnd to form an virtual address; from this address a word containing a 40-bit address is fetched to form a virtual address into any page. This is indirect addressing FROM the process' page, INTO any page.

3.5.5 %@%Opnd - 101

(Opnd1 ONLY)

```
Sub    %@%123,loc22
```

Operand := FETCH(FETCH (0 Segment + Opnd))

Opnd is catenated with the zero page SID to form a virtual address; from this address in the zero page a word containing a 40-bit address is fetched. This virtual address is used to fetch data in any page. This is indirect addressing FROM the zero page, INTO any page.

3.5.6 @%Opnd - 110

(Opnd1 ONLY)

```
Sub    @%123,loc22
```

Operand := FETCH(OSID + FETCH (0 Segment + Opnd))

Opnd is catenated with the zero page SID to form an virtual address. From that address, a 24-bit offset is fetched. This offset is catenated with the OSID to form the virtual operand address. This is indirect addressing FROM the zero page, INTO the process' page. (We do not see a great need for this instruction, however we put it in for symmetry. The compiler (and the compiler writers) can tell us if it is useful.)

3.5.7 <Opnd - 100

(Opnd1 ONLY)

```
Sub    <123,loc22        "note that 123 is subtracted from
                         "the contents of loc22
```

Operand := Opnd

Opnd is a 24-bit immediate operand.

3.6 Conditional Skip

Certain CLOCS instruction include a conditional skip of the next instruction. These instructions are: Add, Sub, Mult, Div, Rem, And, Or, Xor.

For the Add, Sub, Mult, Div, and Rem instructions, the conditions are:

LT result of operation less than zero.

GT result of operation greater than zero.

EQ result of operation equal to zero.

NO result of operation did NOT overflow.

NU result of operation did NOT underflow.

NZ result of operation was NOT a divide-by-zero.

For the And, Or, and XOR instructions, the conditions are:

EQ result of operation equal to zero.

NZ result of operation is not zero.

Each of the conditions correspond to a bit in the instruction. If the bit is set and the condition is true, then the next instruction is not executed. Of course, if no condition is specified, the following instruction will never be skipped.

For ease of generation by the compiler and to ease hand coding assembler, the appropriate possibilities of condition skip have been incorporated into the operation abbreviation. Conditional skips may also be specified by adding the abbreviations above after the required operands. These end of line conditionals override any conditionals specified in the operation code abbreviation, so should be used with care. Following are some example instructions and descriptions of the interpretation and use.

SUBGT Opnd1, Opnd2
Subtract the first operand and from the second operand, placing the result in the second operand, and skip the following instruction if the result was greater than zero. Note that this instruction, when Opnd2 is an immediate one, followed by a trap or branch, provides a p() operation on the semaphore addressed by Opnd1.

DIVNZ Opnd1, Opnd2
Divide operand 1 by operand 2, placing the result in operand 2, and skip the following instruction if the result was NOT a divide-by-zero. If the next instruction is a branch to an error handling routine, this combination allows easy handling of arithmetic exceptions by the user program.

The trap and branch instructions use the same flags, but are conditionally executed instead of conditionally skipping the next instruction.

3.7 Instruction List

In the list of instruction below, the heading identifies the operation and operands for the instruction. In typewriter type, a list of the instruction abbreviations is given, including all abbreviations addressing conditional skips or byte logical operations. In these descriptions Opnd1 and Opnd2 refer to the operand definitions above. The optional conditional description appears at the end of the required operands.

3.7.1 Add Opnd1, Opnd2

```
ADD Opnd1,Opnd2[,Conditional]
ADDGT Opnd1,Opnd2[,Conditional]
ADDGE Opnd1,Opnd2[,Conditional]
ADDEQ Opnd1,Opnd2[,Conditional]
ADDLT Opnd1,Opnd2[,Conditional]
ADDLE Opnd1,Opnd2[,Conditional]
ADDNO Opnd1,Opnd2[,Conditional]
ADDNU Opnd1,Opnd2[,Conditional]
ADDF Opnd1,Opnd2[,Conditional]
```

Operand one and operand two are added in full 64-bit two's complement arithmetic; the result is placed in operand two. (conditional skips: LT, GT, EQ, NO, NU)

3.7.2 Sub Opnd1, Opnd2

```
SUB Opnd1,Opnd2[,Conditional]
SUBGT Opnd1,Opnd2[,Conditional]
SUBGE Opnd1,Opnd2[,Conditional]
SUBEQ Opnd1,Opnd2[,Conditional]
SUBLT Opnd1,Opnd2[,Conditional]
SUBLE Opnd1,Opnd2[,Conditional]
SUBNO Opnd1,Opnd2[,Conditional]
SUBNU Opnd1,Opnd2[,Conditional]
SUBF Opnd1,Opnd2[,Conditional]
```

Operand one is subtracted from operand two; the result is placed in operand two. Arithmetic is full 64-bit two's complement. (conditional skips: LT, GT, EQ, NO, NU)

3.7.3 Mult Opnd1, Opnd2

```
MUL Opnd1,Opnd2[,Conditional]
MULGT Opnd1,Opnd2[,Conditional]
MULGE Opnd1,Opnd2[,Conditional]
MULLT Opnd1,Opnd2[,Conditional]
MULLE Opnd1,Opnd2[,Conditional]
MULEQ Opnd1,Opnd2[,Conditional]
MULNO Opnd1,Opnd2[,Conditional]
MULNU Opnd1,Opnd2[,Conditional]
MULF Opnd1,Opnd2[,Conditional]
```

The low-order 64 bits of operand one are multiplied by the low-order 64 bits of operand two, producing a 64-bit result which is stored in operand two. If the operation results in a number that cannot be represented in 64 bits, an overflow exception will occur. (conditional skips: LT, GT, EQ, NO, NU)

3.7.4 Div Opnd1, Opnd2

```
DIV Opnd1,Opnd2[,Conditional]
DIVGT Opnd1,Opnd2[,Conditional]
DIVGE Opnd1,Opnd2[,Conditional]
DIVEQ Opnd1,Opnd2[,Conditional]
DIVLT Opnd1,Opnd2[,Conditional]
DIVLE Opnd1,Opnd2[,Conditional]
DIVNO Opnd1,Opnd2[,Conditional]
DIVNU Opnd1,Opnd2[,Conditional]
DIVNZ Opnd1,Opnd2[,Conditional]
DIVF Opnd1,Opnd2[,Conditional]
```

Operand two is divided by operand one, and the result is placed in operand two. The operands are 64-bit quantities; the result is a 64-bit quantity. Division is in two's complement integer arithmetic. Division by a divisor greater than the quotient will result in zero. (conditional skips: LT, GT, EQ, NO, NU, NZ)

3.7.5 Rem Opnd1, Opnd2

```
REM Opnd1,Opnd2[,Conditional]
```

Operand two is divided by operand one, and the remainder of this division is placed in operand two. Division is as in the Div instruction. (conditional skips: LT, GT, EQ, NO, NU, NZ)

3.7.6 And Opnd1, Opnd2

```
AND Opnd1,Opnd2[,Conditional]
ANDEQ Opnd1,Opnd2[,Conditional]
ANDNZ Opnd1,Opnd2[,Conditional]
ANDBYT Constant,Opnd1,Opnd2[,Conditional]
ANDBYTEQ Constant,Opnd1,Opnd2[,Conditional]
ANDBYTNZ Constant,Opnd1,Opnd2[,Conditional]
```

Operand two and operand one are ANDed together in bitwise fashion; the result is placed in operand two. fashion, and the result is placed in operand two. These operations are normally bit wise for all bits, but may be applied to only an 8 bit byte selected by a Constant. (conditional skips: EQ, NZ)

3.7.7 Or Opnd1, Opnd2

```
OR Opnd1,Opnd2[,Conditional]
OREQ Opnd1,Opnd2[,Conditional]
ORNZ Opnd1,Opnd2[,Conditional]
ORBYT Constant,Opnd1,Opnd2[,Conditional]
ORBYTEQ Constant,Opnd1,Opnd2[,Conditional]
ORBYTNZ Constant,Opnd1,Opnd2[,Conditional]
```

Operand two and operand one are ORed together in bitwise fashion, and the result is placed in operand two. These operations are normally bit wise for all bits, but may be applied to only an 8 bit byte selected by a Constant. (conditional skips: EQ, NZ)

3.7.8 Xor Opnd1, Opnd2

```
XOR Opnd1,Opnd2[,Conditional]
XOREQ Opnd1,Opnd2[,Conditional]
XORNZ Opnd1,Opnd2[,Conditional]
XORBYT Constant,Opnd1,Opnd2[,Conditional]
XORBYTEQ Constant,Opnd1,Opnd2[,Conditional]
XORBYTNZ Constant,Opnd1,Opnd2[,Conditional]
```

Operand one and operand two are exclusive-ORed together in bitwise fashion, and the result is placed in operand two. These operations are normally bit wise for all bits, but may be applied to only an 8 bit byte selected by a Constant. (conditional skips: EQ, NZ)

3.7.9 Left N, Opnd1, Opnd2

```
LEFT Constant,Opnd1,Opnd2
```

Operand one is shifted left N bits (N is supplied in the flags field of the instruction – it is not a true operand); the result of the shift is placed in operand two. A move is affected by setting N equal to zero. The low-order bits of the result are cleared to zero.

3.7.10 Right N, Opnd1, Opnd2

```
RGHT Constant,Opnd1,Opnd2
```

Operand one is shifted right logically (sign bit is ignored) N bits. The high N bits of the result are cleared to zero. The result is placed in operand two.

3.7.11 RightArith N, Opnd1, Opnd2

```
RGHTA Constant,Opnd1,Opnd2
```

Operand one is right shifted arithmetically (sign extension is performed) N bits. The result is placed in operand two.

3.7.12 Branch Opnd1, Opnd2

```
BRN Opnd1,Opnd2[,Conditional]
BEQ Opnd1,Opnd2[,Conditional]
BLE Opnd1,Opnd2[,Conditional]
BGE Opnd1,Opnd2[,Conditional]
BNE Opnd1,Opnd2[,Conditional]
BGT Opnd1,Opnd2[,Conditional]
BLT Opnd1,Opnd2[,Conditional]
```

The program counter is conditionally loaded from operand one, based on the result of comparisons with operand 2 (LT, GT, EQ, NE, Unconditional, LE, GE). If the addressing mode of operand 1 is such that a new ISEG is fetched, that new ISEG is stored into the MMU. This allows for the changing of instruction contexts. (Operand context is changed via the LoadOSID instruction)

3.7.13 Trap Opnd1, Opnd2

```
TRP Opnd1,Opnd2[,Conditional]
```

Control switches to the context indicated by the trap vector indexed by Opnd1 (a number, not an address), based on the result of comparisons done with operand 2. Comparison conditions are the same as for the branch instruction.

3.7.14 LoadOSID Opnd1

```
LOB Opnd1
```

The OSID from Opnd1 is loaded into the MMU as the primary OSID for this process. If no loaded physical page assigned to the current process has this OSID, the CPU will trap. The operating system may then decide if this is an authorized OSID for the current process.

3.7.15 Floating Point Instructions

Floating point instructions codes are not yet assigned

Floating point instructions are important enough to this machine that we will reserve operation codes for add, subtract, multiply and divide. The instructions will operate on IEEE standard 64 bit floating point numbers with the round to closest rounding option. If not implemented in hardware, the instructions will be executed by the kernel as it handles the unassigned operation code exception. This easily allows addition of floating point hardware later, and the software handling will take maximum advantage of the fast context changes available. Conditional skips shall be handled in the same manner as for fixed point arithmetic.

Chapter 4

Compiler Description

As support for research into issues of architecture and operating systems, a cross compiler for a hypothetical reduced instruction set computer is required. The language to be compiled is C, and the architecture targeted is the CLOCS architecture being designed by Mark Davis and Bill O. Gallmeister.

4.1 Class Project

The CLOCS cross compiler to be built must be a modular system which can be easily modified to output CLOCS assembly language, object code, or high-level simulator constructs. Note that all of these output formats should be roughly isomorphic to one another; producing one from another is mainly a lexical matter.

The CLOCS team has no illusions that the compiler produced by the 240 team will be the final solution to CLOCS' compiler needs; therefore, it is essential that the CLOCS team be able to do work on the compiler after completion of the 240 project. For these reasons, the compiler should be built to output code statistics, and the compiler must be well-structured, well-documented and moderately easy to maintain and modify.

4.2 Fast Context Switch

The CLOCS project is to design a computer architecture to handle real time applications while supporting a full-featured, general purpose operating system. This computer has the ability to change context rapidly.

4.3 CLOCS Architecture

The CLOCS architecture is a simple one. All operations are memory to memory, and the processor has minimal internal state. There are few data types and instructions. Several addressing modes are provided to compensate for the lack of index registers.

4.3.1 Simple Architecture

Because the research is aimed at answering questions regarding the performance of simple machines, and also because this IS a research project, the CLOCS architecture is simple.

RISC

In keeping with the RISC philosophy, CLOCS has few instructions (about twenty), few data types (the word, interpreted as a logical, arithmetic, or addressing entity), and a minimal amount of state. CLOCS has exactly one register, the Status Word.

Research Considerations

Since this is a research machine, a simple architecture was decided upon; this allows us to concentrate more on the central issues of the research.

In addition, to facilitate comparison of CLOCS and currently available designs, it is desired that CLOCS bear some outward resemblance to other existing RISC machines. The machine the CLOCS is designed to resemble the most closely is Sun Microsystems' SPARC processor.

4.3.2 Memory Address Space Organization

16 Meg Directly Addressable

Operand addresses in instructions, as well as the program counter itself, are 24 bits long. This gives a default addressing range of 16 megawords.

One Teraword Total Virtual Address Space

For each process running on the CLOCS cpu, a default segment identifier is supplied for instruction and operands. The segment identifier is prepended to the 24 bit operand or program counter address, to determine the desired address. These segment identifiers are 16 bits long, providing a total of 40 bits of address. Some addressing modes allow altering the

segment identifier,so that all of the virtual address space may be addressed. The address space is 64K segments of 16 megaword each, for 2-to-the-40th (1,099,511,627,776 or one teraword) total virtual address space.

PIDs, Processor Levels and Permissions

CLOCS supports in hardware the notion of distinct processes. In the program status word is the Process Identifier (PID), a 14 bit field that identifies the running process. Associated with each process is a default instruction segment, a default operand (or data) segment and memory access permission for segments being used by the process.

A superuser, PID 0, may access the Memory Management Unit (MMU) registers to establish this information. Any process may read or execute segment 0, and the process with PID 0 may read, write or execute in any segment, but all other memory access must be approved by the MMU. A user process (PID $\neq$ 0) may have permission to read only, read or write, execute only, or read and execute a segment. Segments may be default segments (used when no SID is specified like during instruction fetch or fetching operands using the 24 bit virtual offset in the instruction word).

Memory Scheme

A quick review of terms:

PID (14) Process identifier.

SID (16) Segment identifier.

OSID (16) the default operand SID.

ISID (16) the default instruction SID.

VO (24) Virtual Offset.

PC (24) Program Counter, a VO.

OPND (24) Operand of in an instruction, a VO.

VP (12) Virtual page.

PP (18) Physical page.

PO (12) Physical Offset, the low order bits of VO.

VA (40) Virtual address, SID+VO.

PA (30) Physical address, PP+PO.

The MMU translates the combination of PID, SID, VP to PP and checks the PID's permission on that SID+VP combination. The 30 bit Physical Address gives that machine a real memory capability of 1,073,741,824 words (or 8 gigabytes). All accesses to memory are by 64-bit word access only.

Memory-Mapped Access

CLOCS reduces the variety of its instructions by mapping all state information of the machine into the memory space of the processor. Thus, the State Word, consisting of the PC, PID, and Flags, may be found at location ffff.fffff (This is segment ffff, address fffff). The MMU registers begin at ffff.f00000. Location ffff.feffff contains the number of MMU registers installed on this cpu. Input-output devices are mapped into the memory from ffff.ff0000 to ffff.ffefff. The trap and interrupt vectors, likewise, can be found in the this segment, at addresses ffff.fff000 to ffff.fffffe.

4.3.3 Minimal Processor State

4.3.4 Data Types

CLOCS supports a single arithmetic data type: the 64-bit signed integer represented as a 2's complement. There is provision for an optional data type, a 64 bit IEEE 784 floating point number.

4.3.5 Instruction Format

In this instruction format, the 5 bits of operation code are followed by 5 bits of flags which determine addressing modes for the two operands. The next 6 bits specify flags or a count. The operand, 24 bits long, follows. A number of addressing modes, as described elsewhere in this document, can be applied to the operand(s) by the judicious setting of the addressing mode flags.

4.3.6 Addressing Modes

CLOCS supports seven addressing modes, all of which are available for use with operand one, and four of which are available for operand two. After each addressing mode identification is the bit pattern appearing in the instruction to identify that mode. All modes used for operand 2 have a high order bit of 0, so only the 2 lowest order bits appear in the instruction. In these descriptions, "+" means catenate the two values.

Opnd - 000

(Opnd1, Opnd2)

Operand := FETCH (OSID + Opnd)

OSID, the operand SID, is catenated to the high-order end of Opnd to provide a full

40-bit virtual operand address from which the operand is fetched. This is CLOCS' "direct mode" of addressing.

@Opnd - 001

(Opnd1, Opnd2)

Operand := FETCH (OSID + FETCH(OSID + Opnd))

OSID is catenated to Opnd to form an virtual address. From this address is fetched a 24-bit offset. This offset is catenated with OSID to form the virtual operand address. This is CLOCS' "indirect mode" addressing.

zOpnd - 010

(Opnd1, Opnd2)

Operand := FETCH (0 Segment + Opnd)

The operand is catenated to a SID of zero to arrive at the virtual operand address. This provides rapid zero-page addressing, but otherwise is identical to Direct Addressing.

z@Opnd - 011

(Opnd1, Opnd2)

Operand := FETCH(FETCH (OSID + Opnd))

OSID is catenated to Opnd to form an virtual address; from this address a word containing a 40-bit address is fetched to form a virtual address into any page. This is indirect addressing FROM the process' page, INTO any page.

z@zOpnd - 101

(Opnd1 ONLY)

Operand := FETCH(FETCH (0 Segment + Opnd))

Opnd is catenated with the zero page SID to form a virtual address; from this address in the zero page a word containing a 40-bit address is fetched. This virtual address is used to fetch data in any page. This is indirect addressing FROM the zero page, INTO any page.

@zOpnd - 110

(Opnd1 ONLY)

Operand := FETCH(OSID + FETCH (0 Segment + Opnd))

Opnd is catenated with the zero page SID to form an virtual address. From that address, a 24-bit offset is fetched. This offset is catenated with the OSID to form the virtual operand address. This is indirect addressing FROM the zero page, INTO the process' page. (We do not see a great need for this instruction, however we put it in for symmetry. The compiler (and the compiler writers) can tell us if it is useful.)

<Opnd - 100

(Opnd1 ONLY)

Operand := Opnd

Opnd is a 24-bit immediate operand.

4.3.7 Conditional Skip

Certain CLOCS instruction include a conditional skip of the next instruction. These instructions are: Add, Sub, Mult, Div, Rem, And, Or, Xor.

For the Add, Sub, Mult, Div, and Rem instructions, the conditions are:

LT result of operation less than zero.

GT result of operation greater than zero.

EQ result of operation equal to zero.

NO result of operation did NOT overflow.

NU result of operation did NOT underflow.

NZ result of operation was NOT a divide-by-zero.

For the And, Or, and XOR instructions, the conditions are:

EQ result of operation equal to zero.

NZ result of operation is not zero.

Each of the conditions correspond to a bit in the instruction. If the bit is set and the condition is true, then the next instruction is not executed. Of course, if no condition is specified, the following instruction will never be skipped.

Following are some example instructions and descriptions of the interpretation and use.

SUB GT Opnd1, Opnd2
Subtract the first operand and from the second operand, placing the result in the second operand, and skip the following instruction if the result was greater than zero. Note that this instruction, when Opnd2 is an immediate one, followed by a trap or branch, provides a p() operation on the semaphore addressed by Opnd1.

DIV NZ Opnd1, Opnd2
Divide operand 1 by operand 2, placing the result in operand 2, and skip the following instruction if the result was NOT a divide-by-zero. If the next instruction is a branch to an error handling routine, this combination allows easy handling of arithmetic exceptions by the user program.

The trap and branch instructions use the same flags, but are conditionally executed instead of conditionally skipping the next instruction.

4.3.8 Instruction List

Add Opnd1, Opnd2

Operand one and operand two are added in full 64-bit two's complement arithmetic; the result is placed in operand two. (conditional skips)

Sub Opnd1, Opnd2

Operand one is subtracted from operand two; the result is placed in operand two. Arithmetic is full 64-bit two's complement. (conditional skips)

Mult Opnd1, Opnd2

The low-order 64 bits of operand one are multiplied by the low-order 64 bits of operand two, producing a 64-bit result which is stored in operand two. If the operation results in a number that cannot be represented in 64 bits, an overflow exception will occur. (conditional skips)

Div Opnd1, Opnd2

Operand two is divided by operand one, and the result is placed in operand two. The operands are 64-bit quantities; the result is a 64-bit quantity. Division is in two's complement integer arithmetic. Division by a divisor greater than the quotient will result in zero. (conditional skips)

Rem Opnd1, Opnd2

Operand two is divided by operand one, and the remainder of this division is placed in operand two. Division is as in the Div instruction. (conditional skips)

And Opnd1, Opnd2

Operand two and operand one are ANDed together in bitwise fashion; the result is placed in operand two. (conditional skips)

Or Opnd1, Opnd2

Operand two and operand one are ORed together in bitwise fashion, and the result is placed in operand two. (conditional skips)

Xor Opnd1, Opnd2

Operand one and operand two are exclusive-ORed together in bitwise fashion, and the result is placed in operand two. (conditional skips)

Left N, Opnd1, Opnd2

Operand one is shifted left N bits (N is supplied in the flags field of the instruction - it is not a true operand); the result of the shift is placed in operand two. A move is affected by setting N equal to zero. The low-order bits of the result are cleared to zero.

Right N, Opnd1, Opnd2

Operand one is shifted right logically (sign bit is ignored) N bits. The high N bits of the result are cleared to zero. The result is placed in operand two.

RightArith N, Opnd1, Opnd2

Operand one is right shifted arithmetically (sign extension is performed) N bits. The result is placed in operand two.

Branch Opnd1, Opnd2

The program counter is conditionally loaded from operand one, based on the result of comparisons with operand 2 (LT, GT, EQ, NE, Unconditional, LE, GE). If the addressing mode of operand 1 is such that a new ISEG is fetched, that new ISEG is stored into the MMU. This allows for the changing of instruction contexts. (Operand context is changed via the LoadOBase instruction)

Trap Opnd1, Opnd2

Control switches to the context indicated by the trap vector indexed by Opnd1 (a number, not an address), based on the result of comparisons done with operand 2. Comparison conditions are the same as for the branch instruction.

LoadOSID Opnd1

The OSID from Opnd1 is loaded into the MMU as the primary OSID for this process. If no loaded physical page assigned to the current process has this OSID, the cpu will trap. The operating system may then decide if this is an authorized OSID for the current process.

Floating Point Instructions

Floating point instructions are important enough to this machine that we will reserve operation codes for add, subtract, multiply and divide. The instructions will operate on IEEE standard 64 bit floating point numbers with the round to closest rounding option. If not implemented in hardware, the instructions will be executed by the kernel as it handles the unassigned operation code exception. This easily allows addition of floating point hardware later, and the software handling will take maximum advantage of the fast context changes available.

Bibliography

[1] Harvey M. Deitel. *An Introduction to Operating Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[2] Karsten Schwan, Tom Bihari, Bruce W. Weide, and Gregor Taulbee. High-Performance Operating System Primitives for Robotics and Real-Time Control Systems. *ACM Transactions on Computer Systems*, 5(3):189–231, August 1987.