# Lecture V
# Toy Hardware and Operating System

## §1. Introduction

For use in our OS projects, we introduce "THE Machine" where "THE" is an acronym[1] for **Toy HardwarE**. We also introduce an associated **Toy Operating System** (TOS). Both THE Machine and Toy OS are implemented as C programs. The assembly language for THE Machine is called **Simulated Toy Machine Language** (STML).

The Toy OS executes a single STML program at a time. Sample STML programs will be provided. All our OS projects are based on THE/TOS. One of your first assignment is to modify the Toy OS is to introduce multi-programming.

The source files for THE/TOS are found in a CVS repository which you can get access to. You are to store your projects in the CVS repository, which will be team-based. In order to carry out these projects, you will learn need to learn about using the CVS tool for developing software projects.

## §2. THE Machine

- Hardware Specification: THE Machine has 1 MB of RAM, and this is divided up into 256K of words where each word is 4 bytes. We can address individual words in the RAM. Hence, the physical address space is 18 bits.

  There are 16 registers, named R0, R1,..., R15. The register R0 is the PC (program counter).

  There are three special registers to support multiprogramming: a base register, a limit register, and an program status word (a.k.a. error status flag).

- There are 16 machine instructions. Thus each instruction can be represented by a 4 bit number. These 4-bit numbers are called **OpCodes**. The operands of each instruction consists of at most one 18-bit address (AD), and up to four registers (RA, RB, RC, RD).

- The following table describes these 16 instructions.

[1] THE is the namesake of a multi-tasking OS designed in 1965–6 by the pioneer Dutch Computer Scientist, Edsger Dijkstra. "THE" is an abbreviation of "Technische Hogeschool Eindhoven", now the Eindhoven University of Technology.

Table of STML Instructions

| OpCode | Operation | Operands | Meaning |
|---|---|---|---|
| 0 | LOA | RA, AD | Load contents at location AD to RA |
| 1 | STO | RA, AD | Store contents of RA to location AD |
| 2 | CPR | RA, RB | Copy RB into RA |
| 3 | LOI | RA, RB | RB holds the address of AD. Load the value of AD into RA. |
| 4 | STI | RA, RB | RA holds the address of AD. Store the value in RB to AD. |
| 5 | ADD | RA, RB, RC | RC = RA + RB |
| 6 | SUB | RA, RB, RC | RC = RA - RB |
| 7 | MUL | RA, RB, RC | RC = RA * RB |
| 8 | DIV | RA, RB, RC, RD | RC = RA / RB, and RD = RA % RB. |
| 9 | ICR | RA | RA++ |
| 10 | DCR | RA | RA-- |
| 11 | GTR | RA, RB, RC | RC = (RA > RB) |
| 12 | JMP | RA, AD | Jump to AD. (RA is unused). |
| 13 | IFZ | RA, AD | If (RA == 0) then goto AD. |
| 14 | JMI | RA | RA holds the address of AD. Jump to AD. |
| 15 | TRP | | Trap to the kernel. |

- Remark: the "I" in the instruction names `LOI/STI/JMI` suggests Indirect Addressing, i.e., when the address is stored in a register or location, rather than explicitly given. But the "I" in `ICR` and `IFZ` is just coincidence.
- Each STML instruction can fit into a machine word: the OpCode is laid out in the lowest order 4 bits, followed by the arguments. This layout is illustrated in Figure 1. Here, bit 0 is the lowest-order bit and bit 31 is the highest-order bit.

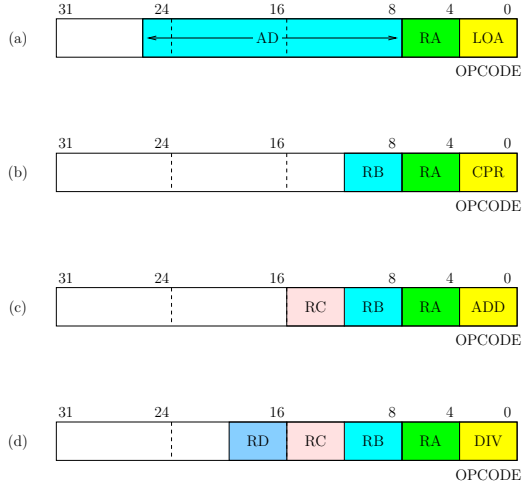


Figure 1: Formats for STML instructions

- For instance, in Figure 1(a), we see that the instruction

  ```
  LOA RA, AD
  ```

  uses bits 0–3 to specify the OpCode, bits 4–7 to specify the registor RA, and bits 8–25 to specify the address AD. This uses up 4+4+18=26 bits. The remaining bits 26–31 are unused.

  Let us suppose that `RA=3` and `AD=10`. Then the bit pattern for this instruction would be

  $$(0000, 0000, 0000, 0000, 0000, 1010, 0011, 0000). \tag{1}$$

- There are 6 distinct instruction formats, depending on OpCode. These formats use at most 26 bits, and as few as 4 bits. Besides the four formats in Figure 1, there are two others. (What are they?)

- We must discuss the TRAP instruction. Register 15 and possibly 14 and 13 will be used. The value of R15 indicates the nature of the system call:

  - R15 = 0. Terminate. The process halts.
  - R15 = 1. Integer Input. Read an integer from standard input. The value of the integer read is returned to the process in register R14. R13 is returned as 1 if an integer has been read and 0 if the end-of-file is reached. Any non-integer value in standard input causes an error.
  - R15 = 2. Integer Output. Print on a new line to standard output in decimal the value of the integer held in register R14.
  - R15 > 2. Future projects will involve defining other trap codes, for forking and for manipulating semaphores, etc.

## §3. The Assembly Language STML

- STML (Simulated Toy Machine Language) is the name of assembly language for THE Machine.

- An STML source file is a text (ASCII) file that normally has the ".stm" extension. It has the following structure:

  - Line 1: name of process
  - Line 2: an integer, specifying the size of memory (in number of words) for this process. This size includes space for code, data, variables, etc.
  - Line 3 onwards: 1 instruction or data item per line. Any line not beginning with a number is ignored. Any item beyond the first number in a line is ignored.

- Let us discuss the ASCII encoding for STML instructions. We view an STML instruction an 32-bit word, which can be intepreted as an integer. By convention, we set the unused bits to zero. With this convention, each instruction is uniquely associated with a 32-bit integer. It is best to write this integer in hexadecimal notation, using the 16 "hexadecimal digits"

  $$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, \mathtt{A}, \mathtt{B}, \mathtt{C}, \mathtt{D}, \mathtt{E}, \mathtt{F}$$

  We normally prefix a hexadecimal notation with the sequence `0x`, to distinguish it from the standard decimal notation (which is unadorned). Thus, `0xA` is 10 (in decimal) while `0x10` is 16. Again, `0xF` is 15 is decimal, and `0x1F` is $16 + 15 = 31$.

  Our main use of hexadecimal notation is, however, to denote STML instructions in a convenient way. For instance, the bit pattern in (1) can now be simply written as `0xA30`. The following table gives more illustrations:

| Hexadecimal Notation | Instruction | Comments |
|---|---|---|
| `0xF` | `TRP` | Trap instruction, no explicit arguments |
| `0xA9` | `INC R10` | Increment register 10 |
| `0xE22` | `CPR R2 R14` | Copy into register 2 the contents of register 14 |

- Let us go through a sample stm code, "fraction.stm".

```
fraction
33

% This program reads four integers A,B,C,D from standard input
% where A < B != 0.  It prints in the standard output,
% successively, the first D "digits" of A/B in base C.
% E.g., if (A,B,C,D)=(1,3,10,5), it prints out "3 3 3 3 3"
% This code is from Professor Davies.

0x1CF0     0  LOA R15 ONE
0xF        1  TRP           --- Read A
0xE12      2  CPR R1 R14    ---   and save in R1
0xF        3  TRP           --- Read B
0xE22      4  CPR R2 R14    ---   and save in R2
0xF        5  TRP           --- Read C
0xE32      6  CPR R3 R14    ---   and save in R3
0xF        7  TRP           --- Read D
0xE42      8  CPR R4 R14    ---   and save in R4
0x512B     9  GTR R2 R1 R5  --- If A > B then fail.
0x195D     A  IFZ R5 EXIT
0x192D     B  IFZ R2 EXIT   --- If B == 0 then fail.
0x1BA0     C  LOA R10 ZERO  --- R10 is the counter for the number of digits.
```

```
0x5A3B     D  GTR R3 R10 R5 --- If C <= 0 then fail.
0x195D     D  IFZ R5 EXIT
0x1DF0     E  LOA R15 TWO   --- The next series of traps will be prints.
          LOOP
0x5137    10  MUL R3 R1 R5  --- print (A*C)/B;
0x76258   11  DIV R5 R2 R6 R7
0x6E2     12  CPR R14 R6
0xF       13  TRP
0x712     14  CPR R1 R7     --- A := (A*C) % B;
0xA9      15  INC R10       --- Increment digit counter and compare to D.
0x5A46    16  SUB R4 R10 R5
0x195D    17  IFZ R5 EXIT
0x100C    18  JMP LOOP
          EXIT
0x1BF0    19  LOA R15 ZERO
0xF       1A  TRP
0         1B  ZERO          --- this is essentially the beginning of the
1         1C  ONE           --- data segment!
2         1D  TWO           --- 1D (=29 in decimal) is last address
```

- Remarks about this code:
  - Although all the instructions are in hexadecimal notation (indicated by the prefix `0x`), we also enter decimal numbers when convenient.

    For instance, line 2 indicates that the program memory size is 33 in decimal. Also, the last three lines of program contains just constants (ZERO, ONE, TWO) written in decimal notation.
  - Following each STML instructions, on the same line, we enter three items of information in human-readable form. E.g., `0xF 3 TRP --- Read B`.
    - STML instruction — `0xF`.
    - Line number — this is in hexadecimal notation, even though we omit the `0x` prefix.
    - The corresponding instructions in "assembly language" — TRP
    - Any comments — this trap corresponds to reading input B.
  - Note that lines `1B, 1C, 1D` are data sections (storing the constants $0, 1, 2$).
  - Can you see why we use 33 words for this process?

## §4. The Toy OS (TOS)

We now describe the Toy OS for THE machine.

- The basic operation of THE/TOS is simply this:

  - It reads a STML file and loads the program into memory (sequentially).
  - All addresses are **relative**: the physical address is equal to the base register plus the relative address.
  - It starts execution of the program starting at relative address 0.

- The compiled version the THE/TOS is called `tos`. To invoke `tos`, you must give it a STML program to execute. Three other optional arguments can be given under the following flags:
  - `-b` $n$: Here $n$ is a non-negative integer. This means that the program will be loaded starting at the base address of $n$. Default value is $n = 0$.
  - `-d` $n$: Here $n$ is 0, 1 or 2. It indicates the debugging level. The default level is $n = 0$, indicating no debugging messages. At higher levels, the Toy OS is increasingly verbose, printing out various diagnostic messages.
  - `-m` $n$: Here $m$ is a positive integer, indicating the maximum number of instructions in the STML program to be executed. This might be useful to avoid infinite loops while you are still debugging a STML program.

  Example: to run a program called `primes.stm` for at most 99 instructions, and at debugging level 1, you type

```
% tos -b 1 -m 99 primes.stm
```

- Program Status Word (PSW) and Error Status: THE Machine does not implement a PSW but has a simpler error flag for the following conditions:
  - Arithmetic errors: divide by 0 (`ERR_DIV_BY_0`) or overflow (`ERR_OVERFLOW`).
  - Address errors (`ERR_ADDR`): address not in partition.
  - PC increment error (`ERR_PC`): incrementing the PC goes outside the partition.
  - I/O error: non-integer in input (`ERR_INPUT`), or reading past End-of-File (`ERR_OUTPUT`).

- THE/TOS is useful as a pedagogical artifact. Nevertheless, it can be made more realistic in several ways.
  - There is no multiprogramming in the basic TOS.
  - There is hardware support for multiprogramming.
  - Having hardware support for a stack pointer would make it easily to implement recursion.

– THE Machine does not model I/O devices which operates asynchronously from the CPU. Thus timing is unrealistic. We would need to simulate I/O blocking and interrupts.

– The OS is not resident in memory. This seems hardest to overcome.

Some of these shortcomings can be overcome in class projects.

## §5. Procedure Calls

We want to extend TOS to support function calls. The standard way to do this is to exploit a stack. We will need a new register SP (stack pointer) to point to the top of this stack.

Sticking to simple THE Machine architecture, we cannot add any new registers to our current instruction set. So we propose to add the SP through the Toy OS, and to implement new traps to access this register.

We will introduce six basic traps which are named

```
CALL, RET, PUSH, POP, GETSP, SETSP.
```

These traps are grouped into three pairs: CALL and RET are for calling and returning from procedures. PUSH and POP is for adding and removing an entry from the stack. Finally, GETSP and SETSP is for moving information from memory to the stack, or vice-versa. The reason why GETSP and SETSP are necessary is because SP is an implicit register which we cannot directly manipulate except through traps.

These traps are viewed as user services, and we would like to distinguish them from traps that perform system services. By definition, system traps will automatically cause a context switch, but user traps do not. So we introduce a convention for our trap vector. Traps 0 to 19 will be considered system traps. Traps 20 and higher will be considered user traps.

In order to understand the way the stack works below, we must remember a convention about how stacks are organize in the memory allocated to a process.

FIGURE: mem[0]... mem[LIMIT-1]

We think of the process memory as starting from address 0 to some upper limit. Let mem[0.. LIMIT-1] denote this sequence of memory location. The executable code of the process is loaded starting at mem[0]. After this, we allocate memory for static variables, and also space for the heap (used in dynamic memory allocation). The stack, however, begins at the mem[LIMIT-1], and grows towards mem[0]. In the above figure, we draw mem[0] at the top, and mem[LIMIT-1] at the bottom. In this sense, the stack is growing upwards.

When we load a process, we assume that SP is initialized to the value LIMIT. To push a value X on the stack, we first **decrement** SP and then set mem[SP]=X. Now, we are ready to describe the four new traps:

- TRAP 20: As usual, R15 contains the constant 20 but R14 will contain either the number of a register (0-15) or an address (>15). In case of a

register, we assume that the register contains an address. In either case, let the address be denoted ADDR. We expect ADDR to be the address or entry point into a procedure. We will refer to this trap call as

`CALL ADDR`.

In fact, our assembler can easily be extended to accept this new syntax. The result of this trap is that:

- Increment register R0 (the PC). Note this incremented value of R0 is the return address after the procedure call.
- Decrement the stack pointer SP.
- Store the value of R0 into mem[SP].
- Set R0 to the value ADDR. This means we jumped to ADDR for our next instruction.

- TRAP 21: R15 contains the constant 21 and R14 contains some (small) non-negative integer N. We will refer to this trap as

  `RET N`.

  The result of this trap is the sequence of actions:

  - Set register R0 to the mem[SP]. Recall that mem[SP] holds the return address after the procedure call.
  - Increment the stack pointer SP.
  - Further, add `N` to SP.

  For now, assume `N` is 0. We will shortly why it is useful to have positive values for `N`.

- TRAP 22: R15 contains the constant 22 but R14 will contain an address ADDR. In order for this trap to be refer to registers we make a convention: when ADDR is in the range 0-15, mem[ADDR] is actually referring to the registers!

  We will refer to this trap as

  `PUSH ADDR`.

  The result of this trap is the following actions:

  - Decrement the stack pointer SP.
  - Set mem[SP] to mem[ADDR].

- TRAP 23: R15 contains the constant 23. R14 will contain an address ADDR. Again, if ADDR is in the range 0-15, mem[ADDR] refer to registers. We will refer to this trap as

  `POP ADDR`.

  The result of this trap is the following actions:

  - Set mem[ADDR] to mem[SP].
  - Increment the stack pointer SP.

- TRAP 24: R15 contains constant 24. R14 contains an ADDR, and R13 contains an offset N. This trap is known

  `GETSP N, ADDR`

  and its effect is to set mem[ADDR] = mem[SP+n]. As usual, if ADDR is between 0 and 15, we are talking about registers.

- TRAP 25: R15 contains constant 25. R14 contains an ADDR, and R13 contains an offset N. This trap is known

  `SETSP N, ADDR`

  and its effect is to set mem[SP+n] = mem[ADDR].

Let now see how to use these traps.

Suppose your procedure uses local variables. We allocate space for these local variables, on the stack. First assume that the procedure takes it arguments in registers.

- Invoke the procedure by executing CALL ADDR.
- Save on the stack the values of any registers that might be used. Use PUSH to do this.
- Allocate space for local variables by decrementing SP.
- Initiatize these local variables.
- Start executing the procedure.

  In this procedure, whenever you need to refer to local variables, use mem[SP+n] for suitable values of n.
- Cleaning up. First, deallocate the space for local variables (increment SP)
- Restore values of registers from the stack. Use POP to do this.
- Return to the caller by executing RET 0.

Passing arguments using registers is fine for most purposes. But there are situations where we want to pass arguments to a procedure "directly", by giving the address of the argument. Another reason might be that we need to pass more arguments than there are registers.

The method is as follows: the caller will push the address of the arguments directly onto the stack before calling the procedure. Upon return, the procedure will do the courtesy of popping the arguments off the stack by calling `RET N` for some positive value of `N`.

APPLICATIONS: Using these function calls, we can now implement libraries. The OS can provide a list of standard libraries. We can keep all library files in a special directory that the OS knows about. When called, it searches this library and loads it. Such libraries are loaded in System space.

## §6. Files

To extend TOS to support file I/O, we introduce new traps.

Exercises

**Exercise 6.1:** What not allow TRAP's to take arguments? Design a trap instruction of the form `TRAP R, ADDR` where R is a register and ADDR an address. ♢

**Exercise 6.2:** Write a STML program that takes as input three numbers $a, b, n$ and it prints the Fibonacci number $f_n$ where the Fibonacci sequence $\{f_i : i \geq 0\}$ is defined by $f_0 = a, f_1 = b$ and $f_{i+1} = f_{i-1} + f_i$. ♢

**Exercise 6.3:** The current I/O traps for THE/TOS only handle integers. We would like to be able to handle general string I/O. To do this, we enhance the original TRP 1 and TRP 2 as follows: Besides Registers 15, 14 and 13, we will now use Register 12. The value of R15 indicates the nature of the system call:

- R15 = 1. Input.
  If R12=0, then we read an integer from standard input, as before. The value of the read integer read returned to the process in register R14. R13 is returned as 1 if an integer has been read, and 0 if the end-of-file is reached. Any non-integer value in standard input causes an error and R13 would get a negative number.
  If R12¿0, then we would read up as many characters as the value of R12 indicates. There would be a buffer reserved in storage, starting at the address stored in R14. After the reading in the characters, the number of read characters would be stored in R13.
  NOTE: this new TRP 1 is consistent with the old TRP because all you need to do to modify an old program to the new one is to load 0 into R12.
- R15 = 2. Output.
  If R12=0, then we will print on the screen the value of the integer held in R14. if R12¿0, then we will print on the screen up to R12 many ASCII characters. The string is stored in a buffer whose starting address is stored in R14.

♢

**Exercise 6.4:** Implement the following six new **user** traps to support function calls, based on the (hidden) register SP or Stack Pointer. ♢

**Exercise 6.5:** (a) Re-write the STML programs in our collection (sort.stm, primes.stm, etc) so that these programs can be called a subroutines.
(b) We want to implement a library system using the function call traps. Modify TOS to support this. ♢

**Exercise 6.6:** Write a recursive version of fib.stm using these new function call traps. ♢

**Exercise 6.7:** The OS should be able to run a batch system.

We can place jobs in files in a special directory, which the system checks.

Certain jobs can be run periodically, or can be specified to start at a particular time.

For batch jobs, we need to arrange I/O to be done through files (via redirection of the standard I/O). ♢

End Exercises