XSM
eXperimental String Machine
Version 1.0

Dr. K. Muralikrishnan
kmurali@nitc.ac.in
NIT Calicut

September 12, 2012

Contents

1 Introduction

1.1 Brief Machine Description

The machine simulator is known as Experimental String Machine (XSM). It is an interrupt driven uniprocessor machine. The machine handles data as strings. A string is a sequence of characters terminated by '\0'. The length of a string is atmost 16 characters including '\0'. The machine interprets a single character also as a string.

1.2 Components of the Machine

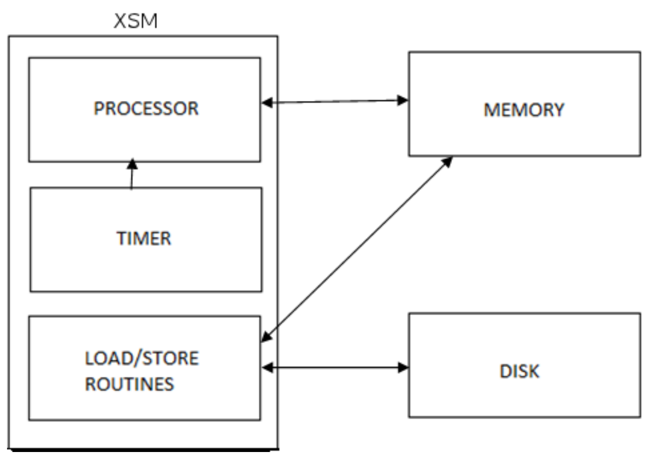

Figure 1: Components of the Machine

- Disk : It is a non-volatile storage that stores user programs (executables) and data files.
- Memory : It is a volatile storage that stores the programs to be run on the machine as well as the operating system that manages the various programs.
- Processor : It is the main computational unit that is used to execute the instructions.
- Timer : It is a device that interrupts the processor after a pre-defined specific time interval.

- Load/Store : It is a macro that performs the functionalities of DMA (Direct Memory Access) controller.

2 Registers

2.1 Introduction

The XSM architecture maintains 24 string registers.

2.2 Register Set

There are 16 General Purpose Registers (GPR), R0 - R15, of which R0 - R7 are Program Registers and R8 - R15 are Kernel Registers. There are 4 temporary registers T0 - T3 which are reserved for code translation. These registers cannot be used by the system programmer. In addition to these 20 registers there are 4 Special Purpose Registers(SPR) namely BP (Base Pointer), SP (Stack Pointer), PID (Process Identifier) and IP (Instruction Pointer).

Name	Register
Program Register	R0-R7
Kernel Register	R8-R15
Temporary Registers	T0-T3
Base Pointer	BP
Instruction Pointer	IP
Stack Pointer	SP
Process Identifier	PID

3 Memory

3.1 Introduction

- The basic unit of memory in XSM is a 16 character string.
- The machine memory can be thought of as a linear sequence of strings.
- A collection of 512 contiguous strings is known as a page.
- The total size of the memory is 64 pages or 32768 (512×64) strings.

- Each string in the memory is identified by the *string address* in the range 0 to 32767 (512×64 - 1). Similarly, each page in the memory is identified by the *page number* in the range 0 to 63.

3.2 Address Translation

There are two kinds of memory addresses,

- Logical address : When a process runs, CPU generates address for the data accessed by this process. This address is called the Logical address.
- Physical address : It is the actual location of the data in the main memory.

Address translation is the process of obtaining the physical address from the logical address. It is done by the machine in the following way.

1. The logical address generated by the CPU is divided by the page size (512) to get the **logical page number**. The remainder is the **offset** of the data within that page.
2. A **page table** is used for address translation. It contains physical page numbers corresponding to each logical page number. The logical page number is used to index the page table to get the corresponding physical page number.
3. The **offset** is then used to refer to the word in the physical page containing the data.

4 Disk Storage

Block : It is the basic unit of storage in the disk.

The disk can be thought of as consisting of a linear sequence of 512 blocks. The size of each block is equal to that of a page in the memory (512 strings).

Any particular block in the disk is addressed by the corresponding number in the sequence 0 to 511 known as the **block number**.

5 Instructions

5.1 Introduction

Every instruction in XSM is 2 strings long. The instructions provided by the XSM architecture can be classified into privileged and unprivileged instructions.

5.2 Classification

5.2.1 Unprivileged Instructions

1. MOV

   - Immediate Addressing:
     Syntax : `MOV Ri, NUM/STRING`
     Copies the `NUM/STRING` to the register `Ri`.
   - Register Addressing:
     Syntax : `MOV Ri, Rj`
     Copies the contents of the register `Rj` to `Ri`.
   - Register Indirect Addressing:
     Syntax : `MOV Ri, [Rj]`
     Copy contents of memory location pointed by `Rj` to register `Ri`.
     Syntax : `MOV [Ri], Rj`
     Copy contents of `Rj` to the location whose address is in `Ri`.
   - Direct Addressing:
     Syntax : `MOV [LOC], Rj`
     Copy contents of `Rj` to the memory address `LOC`.
     Syntax : `MOV Rj, [LOC]`
     Copy contents of the memory location `LOC` to the register `Rj`.

2. Arithmetic Instructions

   - `ADD`, `SUB`, `MUL`, `DIV` and `MOD`.
     General Syntax : `OP Ri, Rj`
     The result of `Ri` op `Rj` is stored in `Ri`. `Ri` and `Rj` should represent integers. Otherwise no operation is performed on the registers.
   - `INR`
     Syntax : `INR Ri`
     Increments the value of register `Ri` by 1. `Ri` should represent an integer. Otherwise no operation is performed on `Ri`.

- `DCR`
  Syntax : `DCR Ri`
  Decrements the value of register `Ri` by 1. `Ri` should represent an integer. Otherwise no operation is performed on `Ri`.

3. Logical Instructions

   - `LT`
     Syntax : `LT Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is less than that in `Rj`. `Ri` is set to 0 otherwise. Strings can also be compared using `LT`
   - `GT`
     Syntax : `GT Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is greater than that in `Rj`. `Ri` set to 0 otherwise. Strings can also be compared using `GT`
   - `EQ`
     Syntax : `EQ Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is equal to that in `Rj`. Set to 0 otherwise. Strings can also be compared using `EQ`
   - `NE`
     Syntax : `NE Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is not equal to that in `Rj`. Set to 0 otherwise. Strings can also be compared using `NE`
   - `GE`
     Syntax : `GE Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is greater than or equal to that in `Rj`. Set to 0 otherwise. Strings can also be compared using `GE`
   - `LE`
     Syntax : `LE Ri, Rj`
     Stores 1 in `Ri` if the value stored in `Ri` is less than or equal to that in `Rj`. Set to 0 otherwise. Strings can also be compared using `LE`

4. Labels
   Syntax : `LABEL` *`labelname`*
   Creates a label with a *`labelname`* which is a string. Labels are used in branching instruction to specify memory location of target instructions.

5. Branching Instructions

   Branching is achieved by changing the value of the `IP` to the address of a specified `labelname`.

   - `JZ`
     Syntax : `JZ Ri, labelname`
     Jumps to `labelname` if the contents of `Ri` is zero.
   - `JNZ`
     Syntax : `JNZ Ri, labelname`
     Jumps to `labelname` if the contents of `Ri` is not zero.
   - `JMP`
     Syntax : `JMP labelname`
     Unconditional jump to address specified by `labelname`

6. Stack Instructions

   - `PUSH`
     Syntax : `PUSH Ri`
     Increment `SP` by 1 and copy contents of `Ri` to the location pointed to by `SP`.
   - `POP`
     Syntax : `POP Ri`
     Copy contents of the location pointed to by `SP` into `Ri` and decrement `SP` by 1.
     For both these instructions `Ri` may be any register except `IP`.

7. Subroutine Instructions

   The `CALL` instruction copies the address of the next instruction to be fetched (`IP` + 2) on to location `SP` + 1, increments `SP` by one and transfers control to the `labelname` specified. The `RET` instruction restores the `IP` value stored at location pointed by `SP`, decrements `SP` by one and continues execution fetching the next instruction pointed to by `IP`. The subroutine instructions provide a neat mechanism for procedure evocations.

   - `CALL`
     Syntax : `CALL labelname`
     Increment `SP` by 1, transfers `IP`+1 to location pointed to by `SP` and jumps to `LABEL`

- `RET`
  Syntax : `RET`
  Sets `IP` to the value pointed to by `SP` and decrements `SP`.

8. Input/Output Instructions

   - `IN`
     Syntax : `IN Ri`
     Transfers the contents of the standard input to `Ri`.
   - `OUT`
     Syntax : `OUT Ri`
     Transfers the contents of `Ri` to the standard output.

9. `START`
   Syntax : `START`
   `IP` will be initialised to this instruction automatically when a program is taken for execution.

10. `END`
    Syntax : `END`
    This instruction is marks the end of a program.

11. `INT`
    Syntax : `INT num`
    Generates an interrupt to the kernel with `num` as a parameter. (Read Section 6)

5.2.2 Privileged Instructions

There are four privileged instructions. They are:

1. `IRET`
   Syntax : `IRET`

   Set `IP` to the value pointed by `SP` and decrements `SP` by one. `IRET` does the same action as `RET`, but it tells the processor that the interrupt handler has finished. With the execution of the `IRET` instruction, interrupts are enabled. (Read Section 6)

2. `LOAD`
   Syntax : `LOAD` *`pagenum blocknum`*

This instruction loads the block specified by the *blocknum*, from the disk, to the page specified by the *pagenum* in the memory. *blocknum* and *pagenum* should be numbers or registers containing numbers. Otherwise no action is performed.

3. `STORE`
   Syntax : `STORE` *blocknum pagenum*
   This instruction stores the page specified by the *pagenum*, from the memory, to the block specified by the *blocknum* in the disk. *blocknum* and *pagenum* should be numbers or registers containing numbers. Otherwise no action is performed.

4. `HALT`
   Syntax : `HALT`
   This instruction causes the simulator to halt immediately.

5.3 Processor Modes

The XSM architecture is interrupt driven and uses a single processor. There are two modes of operation, the user mode and the kernel mode.

- User mode : Only unprivileged instructions can be executed in this mode.

- Kernel mode : Both privileged and unprivileged instructions can be executed in this mode. The processor comes to know about the mode in which the system is running by looking at the value in the `IP` register.

6 Interrupts

6.1 Introduction

Interrupts are mechanisms by which the user code interrupts the execution of the processor and passes control to the kernel to accomplish low level functionalities like disk access, arithmetic exception handling etc.
Interrupt Service Routine(ISR) : The kernel provides routines to accomplish the functionality for which an interrupt has been generated. These routines are known as Interrupt Service Routines.
Note: Every ISR should end with an `IRET` instruction.

6.2 The `INT` instruction

The instruction used to generate an interrupt is `INT`.
Syntax : `INT n`
The `INT` instruction passes control to the Interrupt Service Routine (ISR) for this interrupt located at the physical address computed using the value n. Address computation is done as follows. The physical address of the ISR corresponding to interrupt number n is given by: Physical Address = (56 + n) Page Size Note that the interrupts are disabled once this instruction is executed, since we do not allow interrupts to occur in kernel mode.

6.3 Types of Interrupts

There are 8 interrupts (numbered from 0 to 7) supported by the XSM architecture. The interrupts 0 and 7 are hardware interrupts and the remaining interrupts (1 to 6) are software interrupts.
Details of hardware interrupts are as follows.

- `INT 0` : This is the timer interrupt which interrupts the processor forcing a context switch. It contains the code for the scheduler of the operating system, which schedules the CPU time among the various active processes. Note that this interrupt cannot be called from the user/kernel mode.
- `INT 7` : It is generated when the following exceptions1 occur:
    1. Illegal memory access : occurs when any address generated by the process does not lie in the range [0, 959].
    2. Arithmetic exception : occurs when divisor is 0.
    3. Illegal instruction : occurs when an attempt is made to execute an instruction not belonging to the instruction set and also when the operands to the instruction is not legal. Eg: `MOV 4 R0`, `MOV IP 4` when executed in user mode. These instructions are considered illegal.
    4. Stack overflow and stack underflow : Stack overflow occurs when the value in the SP register exceeds 959 and stack underflow occurs when the value falls below 768.
- `INT 1`, `INT 2` : These interrupts are used for the various file system calls.

- `INT 3`, `INT 4` : These interrupts are used for the various process system calls.
- `INT 5`, `INT 6` : These interrupts have been kept reserved for future use.The interrupts 1, 2, 3 and 4 are unprivileged and can be called from user mode.