XSM
eXperimental String Machine
Version 1.0

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

October 24, 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'. Each of these strings is stored in a **word** (Refer Section 3). The machine interprets a single character also as a string.

1.2 Components of the Machine

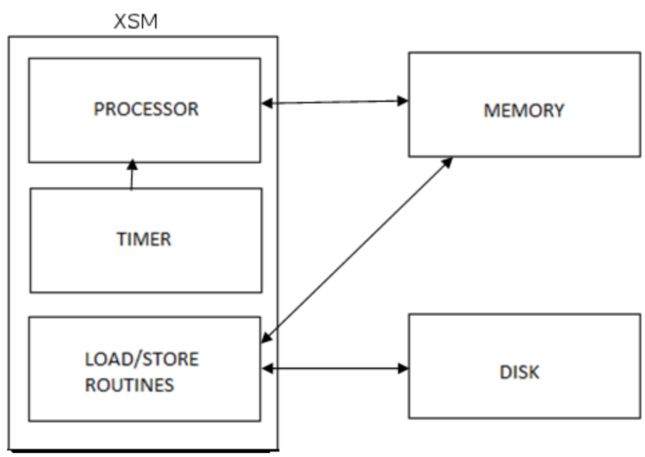

Figure 1: Components of the Machine

- **Disk** : It is a non-volatile storage that stores user programs (executables) and data files. The Operating System code is also stored in the disk.
- **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. (Refer Section 6)

2 Registers

2.1 Introduction

The XSM architecture maintains 26 registers (each one **word**).

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. The registers `T0 - T3` are not intended to be used by the system programmer. In addition to these 20 registers there are 6 Special Purpose Registers(SPR) namely `BP` (Base Pointer), `SP` (Stack Pointer), `PID` (Process Identifier), `IP` (Instruction Pointer), `PTBR` (Page Table Base Register) and `PTLR` (Page Table Length Register).

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
Page Table Base Register	PTBR
Page Table Length Register	PTLR

3 Memory

3.1 Introduction

- The basic unit of memory in XSM is a **word** (length = 16 bytes).
- The machine memory can be thought of as a linear sequence of **words**.

- A collection of 512 contiguous **words** is known as a **page**.
- The total size of the memory is 64 **pages** or 32768 (512×64) **words**.
- Each **word** in the memory is identified by the *word address* in the range 0 to 32767. Similarly, each **page** in the memory is identified by the *page number* in the range 0 to 63.
- The *page number* corresponding to a word is obatained by the formula,

$$page\ number = \lfloor \frac{word\ address}{512} \rfloor$$

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 resides in the memory, the location of which is pointed to by **PTBR** (Page Table Base Register). The number of entries in the page table is stored in **PTLR** (Page Table Length Register). Each entry of the page table is two words long.
   - First word contains physical page number corresponding to a logical page number.
   - The second word indicates if the entry is valid or invalid. A string starting with **1** indicates a valid page table entry.

   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.

The example below shows the address translation correspoding to the logical address 13532.

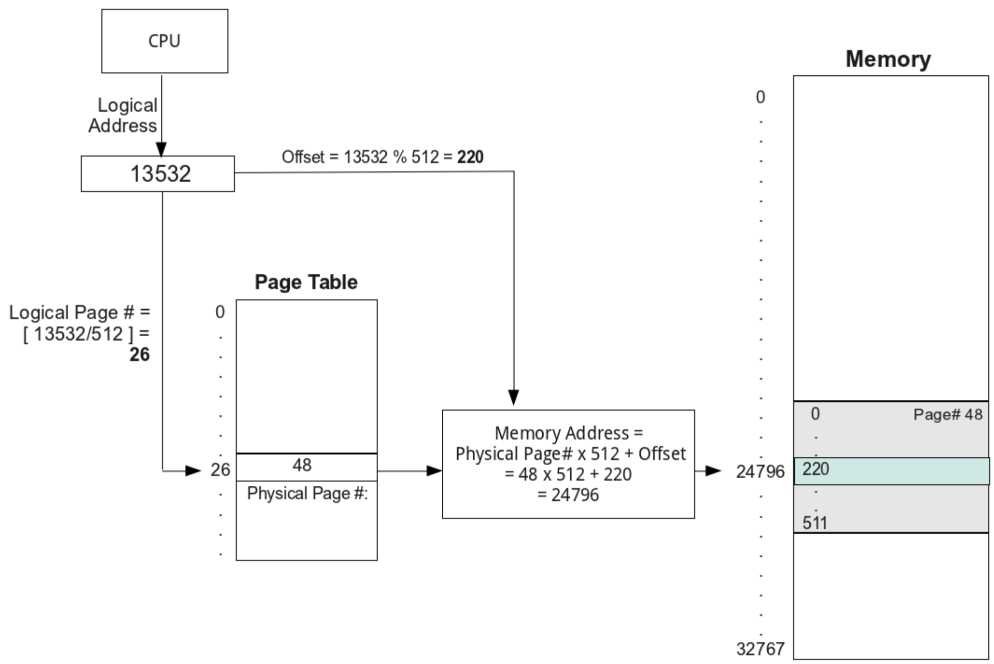

Figure 2: The logical address generated by the CPU is 13532, so the page number is $\lfloor 13532/512 \rfloor = 26$ and offset is 13532 mod 512 = 220. Let the 26^{th} entry in the page table be 48. Thus the resultant physical address is $48 \times 512 + 220 = 24796$.

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 words). The total disk capacity is $512 \times 512 = 262144$ **words**.

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

0 - 511	512 - 1023	...	261632 - 262143
Block 0	Block 1	....	Block 512

Figure 3: Disk Structure

5 Supported Datatypes

XSM supports 2 different datatypes and their operations, namely **Strings** and **Integers**. However in the lowest level data is stored as strings.

5.1 Strings

Strings are sequence of characters which may include alphabets, numerals and special characters. Every string is terminated with a *null character* ('\0'). Operations that can be performed on strings include lexicographic comparisons.

5.2 Integers

Apart from strings, XSM supports integers and its operations. The operations that can be performed on integers include arithmetic operations and comparison operations. A jump can also performed by checking if a register has 0 in it.

6 Instructions

6.1 Introduction

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

6.2 Classification

6.2.1 Unprivileged Instructions

1. `MOV`

   - Register Addressing:
     Syntax : `MOV Ri, Rj`
     Copies the contents of the register `Rj` to `Ri`.
   - Immediate Addressing:
     Syntax : `MOV Ri, INTEGER/STRING` Copies the `INTEGER/STRING` to the register `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`.
   - Direct Indexed Addressing:
     Syntax : `MOV [LOC] Rj, Ri`
     Copy contents of `Ri` to the memory address `LOC` + (value in `Rj`)
     Syntax : `MOV [LOC] Index, Rj`
     Copy contents of `Ri` to the memory address `LOC` + `Index`. `Index` must be an integer value.
     Syntax : `MOV Ri, [LOC] Rj`
     Copy contents in the memory address `LOC` + (value in `Rj`) to the register `Ri`
     Syntax : `MOV Ri, [LOC] Index`
     Copy contents of the memory address `LOC` + `Index` to the register `Ri`. `Index` must be an integer value.

2. **Arithmetic Instructions**
   Arithmetic Instructions perform arithmetic operations on registers containing integers. If the register contains a non-integer value, an excpetion is raised (Refer Section 7.1)

- `ADD`, `SUB`, `MUL`, `DIV` and `MOD`.
  General Syntax : `OP Ri, Rj`
  The result of `Ri` op `Rj` is stored in `Ri`.
- `INR`
  Syntax : `INR Ri`
  Increments the value of register `Ri` by 1.
- `DCR`
  Syntax : `DCR Ri`
  Decrements the value of register `Ri` by 1.

3. **Logical Instructions**
   Logical instructions are used for comparing values in registers. Strings can also be compared according to the lexicographic ordering of ASCII.

   - `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.
   - `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.
   - `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.
   - `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.
   - `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.
   - `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.

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 on to location `SP` + 1. It also increments `SP` by one and transfers control to the `labelname` specified. The address of the instruction to be fetched is in `IP + 2` (each instruction is 2 memory words). 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`
  Increments `SP` by 1, transfers `IP`+2 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. `END`
   Syntax : `END`
   This instruction is marks the end of a program.

10. `INT`
    Syntax : `INT n`
    Generates an interrupt to the kernel with `n` (1 to 6) as a parameter. It also disables the interrupts. (Read Section 7)

6.2.2 Privileged Instructions

There are four privileged instructions. They can only be executed in kernel mode (Refer to 6.3). These instructions 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 7)

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. An excpetion is raised (Refer Section 7.1) for invalid arguments or illegal memory access.

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. An excpetion is raised (Refer Section 7.1) for invalid arguments or illegal memory access.

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

6.3 Privilege Modes

The XSM architecture is interrupt driven and uses a single processor. There are two privilege modes of execution, the user mode and the kernel mode. The privilege mode of execution is determined by the value of IP, i.e. the area in memory where the currently executing code executes.

- User mode : Only unprivileged instructions can be executed in this mode. The value of IP cannot be changed in user mode.

- Kernel mode : Both privileged and unprivileged instructions can be executed in this mode.

Page Number	**Privilege Level**
0	ROM Code
1 - 15	Kernel
16 - 63	User

7 Interrupts

Interrupts are mechanisms by which the machine interrupts the execution of the processor and passes control to the kernel to execute interrupt (or exception) handler code. Interrupts might indicate errors, such as a memory access violation (page fault), a timer interrupt or the OS needs to perform an operation to support a running program, such as a software interrupt.

The process saves its current state before starting execution of the handler, and then resumes the state once the handler finishes its execution. Interrupts are disabled when the interrupt handler code is execuing.

7.1 Exceptions

Exceptions are anomalous situations which changes the normal flow of execution. It is generated when the following events occur:

1. Illegal memory access : occurs when any address generated by the process lies outside its logical address space. The logical page number generated should be between 0 and the value of `PTLR`.
2. Arithmetic exception : occurs when divisor is 0.
3. Illegal operands : occurs when operands contain invalid data corresponding to the instruction.
4. 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.
5. Page Fault : occurs when the page table entry corresponding to the logical address is invalid. Page fault exception interrupts the execution of the CPU and causes a page fault handler routine to be executed by the kernel, which resides in *page number* 7 (address = 3584)

All exceptions other than page fault exception results in termination the machine.

7.2 Timer Interrupt

Timer Interrupt is generated by the machine, usually at set timer intervals. This interrupt cannot be invoked from the user/kernel mode.

It transfers control of execution, i.e. changes the value of IP to *page number* 8 (address = 4096). This is the timer interrupt which interrupts the processor. Generally it is supposed to contain the code for the scheduler of the operating system, which schedules the CPU time among the various active processes.

7.3 Software Interrupts

Software Interrupts interrupts are unprivileged and can be called from user mode. 7 Interrupt instructions are provided by the machine which causes the execution of ISR (Interrupt Service Routines) in 7 different pages in the memory.

7.3.1 The `INT` instruction

The instruction used to generate a software 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. The physical address of the ISR corresponding to interrupt number n is given by:

```
Physical Address = (8 + n) x Page Size
```

Note that the interrupts are disabled once this instruction is executed as interrupts cannot be executed in kernel mode.

The 7 `INT` instructions are :

- `INT 1`
  It transfers control of execution to *page number* 9 (address = 4608)
- `INT 2`
  It transfers control of execution to *page number* 10 (address = 5120)
- `INT 3`
  It transfers control of execution to *page number* 11 (address = 5632)

- `INT 4`
  It transfers control of execution to *page number* 12 (address = 6144)
- `INT 5`
  It transfers control of execution to *page number* 13 (address = 6656)
- `INT 6`
  It transfers control of execution to *page number* 14 (address = 7168)
- `INT 7`
  It transfers control of execution to *page number* 15 (address = 7680).

Brief memory outline for XSM is given below.

Page Number	**Contents**	**Word Address**
0	ROM Code	0 – 511
1	OS Startup Code	512 – 1023
2 – 6	OS Structures	1024 – 3583
7	Page Fault Handler	3584 – 4095
8	Timer Interrupt Routine	4096 – 4607
9	Interrupt 1	4608 – 5119
10	Interrupt 2	5120 – 5631
11	Interrupt 3	5632 – 6143
12	Interrupt 4	6144 – 6655
13	Interrupt 5	6656 – 7167
14	Interrupt 6	7168 – 7679
15	Interrupt 7	7680 – 8191
16 – 63	User Programs	8192 – 32767