Communicating with Environment

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Sketch

- Input and output (I/O)
- Interrupts
- Direct memory access (DMA)
- Exceptions
- System calls

I/O devices

- A computer that can only execute instructions for computing and accessing memory is not very useful

  Environment here refers to the end user.
  - There is no way for the environment to give inputs to the computer or examine outputs of computation
- A typical computer interfaces with a large number of I/O devices
  - Keyboard, display, mouse, speaker, microphone, hard disk, printer, USB devices, etc.
  - Need a mechanism to communicate with these devices
    - How to read a key punch or how to display

I/O devices

- A typical I/O device has a set of command registers and a set of data registers
  - These are assigned unique addresses and can be read or written to through load or store instructions
    - Known as memory-mapped I/O registers
  - To the computer, these appear as memory locations
    - The only difference is that they are not in DRAM, but in I/O devices
  - For example, printing something on the printer involves storing the data to be printed to the printer data registers and an appropriate command to the printer command registers

I/O devices

- Certain I/O commands send responses back to the computer
  - A keyboard read needs to be conveyed to the computer
  - Completion of a disk read needs to be conveyed to the computer
  - Any read operation must be communicated to the computer
  - To detect completion of a disk or keyboard read, one possibility is to continuously poll a register of the disk controller or the keyboard controller
    - Wastes computer’s time (computer could do something else during this time)

I/O devices

- Polling works only if the computer is aware that it will receive some response from a certain I/O device
  - Certain responses are accidental (e.g., ctrl+C to terminate a program or a mouse click)
    - In such cases, the computer did not know beforehand and was not polling the memory-mapped register
- An efficient solution that covers all cases is implemented using interrupts
  - These are signals sent by the I/O devices to the computer
  - For example, a key punch generates an interrupt

Interrupts

- Interrupts stop the normal instruction processing of a computer and make it execute an interrupt handler function
  - Interrupts can be generated by hardware (e.g., I/O devices) or software (e.g., exceptions and system calls)
    - MIPS treats all these as exceptions (any exceptional situation that interrupts normal instruction execution)
  - There are two ways to implement interrupt or exception handlers
    - Vectored interrupts or exceptions
    - Cause-based interrupt handling (non-vectored)

Vectored interrupts

- Reserve an area of memory (outside user memory map) to store interrupt handlers
  - Each interrupt is given a number and an array stores the starting addresses of the interrupt handlers
    - Starting address of interrupt handler i is stored in IV[i]
    - Assume that the interrupt number is in $1 and the starting address of array IV is in $3

```
sll  $2, $1, 2
add  $4, $3, $2
lw   $4, 0($4)
jalr $4
```

Vectored interrupts

- The handlers stored at vector locations are usually very small
  - These handlers further inspect the reason for the interrupt and jump to bigger interrupt service routines (ISRs)
    - For example, all hardware interrupts are usually given a single number and they all call the same interrupt handler
    - The interrupt handler finds out the source device of the hardware interrupt by consulting a special status register that stores the source/cause of interrupt
    - The interrupt handler calls the appropriate ISR

Non-vectored interrupts

- MIPS implements non-vectored interrupts
  - All interrupts jump to the same fixed location (0x80000180)
  - MIPS maintains a status register and a cause register
    - Status register encodes the source of interrupts (six hardware and two software interrupt levels)
    - Cause register encodes the reason for interrupt (used mostly for software interrupts)
  - The interrupt handler examines the status and cause registers and jumps to the appropriate ISR
- ISR copies the input from I/O device data registers (e.g., keyboard buffer) to memory

Direct memory access (DMA)

- Copying large amounts of data from input devices to memory may take a significant amount of time
  - Reading files from disk
- Same applies to copying from memory to output devices
  - Write to files on disk
- Occupying the computer during this time wastes computer's resources
  - Could compute something useful during this time
- Direct memory access (DMA) frees up the computer during data copying from/to I/O

Direct memory access (DMA)

- Computer initializes a specialized hardware called DMA controller by setting up the copy address range and the number of bytes to be copied
- DMA controller does the actual copy operation
  - DMA controller when copying from an input device sends the data to the DRAM controller for writing
  - DMA controller when copying to an output device sends read requests to the DRAM controller
  - When the copying completes, the DMA controller sends an interrupt to the computer to notify about the DMA completion

Exceptions

- Exceptions refer to situations where the running program exhibits unexpected behavior
  - Arithmetic overflow, divide by zero, fetching and decoding an illegal opcode, accessing an illegitimate address or unaligned address (MIPS)
  - In such situations, the PC of the offending instruction is saved in a register called exception PC (EPC) and the program counter is changed to point to a location that has a "trap" instruction
  - The trap instruction allows the computer to enter the operating system which further invokes the appropriate exception handler after examining the cause register

Exceptions

- Some exceptions are restartable while some exceptions are not
  - Restartable exceptions return to normal execution of the program by copying the EPC or EPC+4 (depending on the situation) into a general-purpose register ($X) and issuing a jr $X instruction after the exception handler completes
    - Example: non-availability of code/data in memory (causes a page fault exception)
  - Non-restartable exceptions typically lead to termination of the running program with an appropriate message printed on display
    - Arithmetic exceptions, illegal opcode, crossing legitimate memory boundary, etc.

System calls

- System calls are pseudo-function calls to request access to certain hardware/software resources of the computer system
  - Reading from a file on disk, reading from keyboard, writing to display, writing to a file, allocating dynamic memory, etc. involve system calls
  - Some computers refer to system calls as software interrupts
  - MIPS ISA has the syscall instruction for this purpose
    - R format, has no operand, function 0xc
    - Before invoking the syscall instruction, few registers need to be set up with appropriate information

syscall instruction

- Every system call has a number to indicate the purpose of the system call
  - Reading from a file, writing to a file, allocating dynamic memory all have different system call numbers
  - The system call number must be placed in register $v0 in MIPS
  - A system call can accept four arguments in registers $a0, $a1, $a2, $a3
    - More than four arguments can be passed by packing fourth argument onward in a structure and passing a pointer to the structure in $a3
  - A system call returns any expected response in $v0

syscall instruction

- A syscall instruction executes by first changing the program counter to jump to a location
  - Vectored implementations treat all system calls as software interrupts and all system calls are assigned a fixed location in interrupt vector array
  - Non-vectored implementations jump to the same location for all kinds of interrupts
  - The code in this location further examines $v0 and jumps to the appropriate system call handler
  - System call handler accesses the argument registers and does the necessary things

Examples of system call

- Consider the read system call of UNIX
  - Invoked as part of C library functions scanf, fscanf, read, etc.
    - stdin is treated like a file
  - Three arguments: $a0 should have file descriptor (a non-negative integer representing the file), $a1 should have destination memory buffer address, $a2 should have number of bytes to read
    - Number of bytes to read is inferred from the data types of the arguments of the high-level function call
  - SPIM simulator uses different system calls and conventions for reading from keyboard
    - Only read_string requires buffer address and length in $a0 and $a1; other keyboard reads do not have args

Examples of system call

- Consider the read system call
  - On return of a read system call, $v0 contains the number of bytes read
    - Note that the actual bytes read can be found in memory starting from the address passed in $a1 to the system call
  - SPIM uses a different convention for reading from keyboard: on return of a read system call for reading from keyboard, $v0 or $f0 (depending on type of data read) contains the actual value read (except for read_string, which does not have any return value)

Examples of system call

- Complete path of scanf
  - C program calls scanf function
  - Program jumps to scanf library function (jal)
  - Sets up $v0, $a0, $a1, $a2
  - syscall instruction executes (part of scanf func.)
  - Invokes system call handler for reading from file
  - Syscall handler goes to sleep until interrupted
  - Keyboard punch generates interrupt to computer
  - ISR wakes up system call handler
  - Tail code of system call handler copies data from keyboard buffer to computer memory pointed to by $a1
  - Returns to C program that called scanf

Examples of system call

- Consider the write system call
  - Invoked as part of C library functions printf, fprintf, write, etc.
    - stdout is treated like a file
  - Three arguments: $a0 should have file descriptor (a non-negative integer representing the file), $a1 should have source memory buffer address containing the characters to be written, $a2 should have number of bytes to write
    - Number of bytes to write is inferred from the data types of the arguments of the high-level function call
  - SPIM simulator uses different system calls and conventions for writing to display
    - Only argument is $a0 or $f12 holding the value to be printed ($a0 contains a pointer to string in print_string)

Examples of system call

- Complete path of printf
  - C program calls printf function
  - Program jumps to printf library function (jal)
  - Sets up $v0, $a0, $a1, $a2
  - syscall instruction executes (part of printf func.)
  - Invokes system call handler for writing to file
  - Copies data ($a2 bytes) from source buffer pointed to by $a1 to the display device’s memory
  - Invokes the display device driver to actuate the low-level devices (CRT or LCD or LED) for printing the characters
  - Returns to C program that called printf

Examples of system call

- Consider the sbrk system call used for allocating dynamic memory
  – Invoked as part of C libray function malloc
  – One argument: $a0 should contain the number of bytes to be allocated
  – On return, $v0 contains the starting address of the allocated memory region

Example of syscall in SPIM

- SPIM assembly language program for printing the string "Hello World\n" to display

```
        .data
msg:    .asciiz "Hello World\n"

        .text
        .globl main
main:   li $v0, 4           # syscall 4 (print_str)
        la $a0, msg         # argument: string address
        syscall             # print the string

        jr $ra              # retrun to caller
```

Example of syscall in SPIM

- SPIM assembly language program for reading an integer, adding 42 to it, and printing the result to display

```
        .text
        .globl main
main:   li $v0, 5               # syscall 5 (read_int)
        syscall
        addi $a0, $v0, 42       # print_int argument
        li $v0, 1               # syscall 1 (print_int)
        syscall                 # print the integer

        jr $ra                  # retrun to caller
```

Example of syscall in SPIM

- SPIM assembly language program for reading two floats, adding them, and printing the result to display

```
        .text
        .globl main
main:   li $v0, 6                  # syscall 6 (read_float)
        syscall
        mov.s $f12, $f0            # return value in $f0
        li $v0, 6
        syscall
        add.s $f12, $f0, $f12      # print_float arg
        li $v0, 2                  # syscall 2 (print_float)
        syscall                    # print
        jr $ra                     # retrun to caller
```

Example of syscall in SPIM

- SPIM assembly language program for reading ten integers, storing them in an array, adding them, and printing the result

```
        .data
arrayX: .space 40
msg:    .asciiz "Sum of values: "
endmsg: .asciiz "\n"

        .text
        .globl main
main:   addi $t0, $0, 10      # i ← 10
        la $t1, arrayX        # $t1 ← arrayX
loop:   li $v0, 5             # syscall 5 (read_int)
        syscall
        sw $v0, 0($t1)        # *arrayX ← $v0
        addi $t1, $t1, 4      # arrayX++
        addi $t0, $t0, -1     # i--
        bne $t0, $0, loop
```

Example of syscall in SPIM

```
        addi $t0, $0, 10        # i ← 10
        la $t1, arrayX          # $t1 ← arrayX
        xor $t2, $t2, $t2       # sum ← 0
loop1:  lw $v0, 0($t1)          # $v0 ← *arrayX
        add $t2, $t2, $v0       # sum ← sum + $v0
        addi $t1, $t1, 4        # arrayX++
        addi $t0, $t0, -1       # i--
        bne $t0, $0, loop1
        li $v0, 4               # print_string
        la $a0, msg             # argument: string
        syscall                 # print the string
        li $v0, 1               # print_int
        add $a0, $t2, $0        # int to print
        syscall
        li $v0, 4               # print_string
        la $a0, endmsg          # argument: string
        syscall                 # print the string

        jr $ra                  # retrun to caller
```

System calls supported by SPIM

- print_int: no. 1, $a0 should have int value
- print_float: no. 2, $f12 should have float value
- print_double: no. 3, ($f12, $f13) should have double value
- print_string: no. 4, $a0 should have the pointer to the string
- print_char: no. 11, $a0 should have the ASCII value of char
- sbrk: no. 9, $a0 should have the number of bytes to be allocated, return address in $v0

System calls supported by SPIM

- read_int: no. 5, return int value in $v0
- read_float: no. 6, return float value in $f0
- read_double: no. 7, return double value in ($f0, $f1)
- read_string: no. 8, $a0 should have the destination memory buffer address, $a1 should have the length of the string
- read_char: no. 12, return char value in $v0
- exit: no. 10 (terminates the calling program)
- exit2: no. 17 (terminates spim)

System calls supported by SPIM

- Also support system calls to open a file, read from a file, and write to a file
  - Useful for programs operating on files