Emulating X-Makina's devices

Larry Hughes, PhD
10 July 2018

1 Introduction

In Assignment 2, an emulator is to be designed, implemented, and tested. In addition to the CPU, memory, and bus, the emulator supported devices.

A device is either an input device (data from the outside world is supplied to the CPU) or an input device (the CPU supplies data to the outside world).

Regardless of the devices, all devices have the same structure: a one-word of low memory, referred to as the *device port*, consisting of a control/status register and data register (from the description of devices in assignment 2):

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Data (I/O)								Reserved				OV	DBA	I/O	IE

where:

IE: Interrupt enable (bit 0). Indicates whether this device is enabled for interrupts. Enable by setting the bit. If the bit is not enabled, the device can still be active; however, in order to determine whether a status change has occurred, it is necessary to poll the device. This is a control bit.

I/O: Input/output enable (bit 1). Indicate whether the device is for input (set bit) or output (clear bit). This is a control bit.

DBA: Data-byte or data-buffer available (bit 2). If the device is programmed to be an input device and this bit is set, it means that a data-byte is available for reading from the data register. The bit is cleared by the device when the data register is read by the program. The register remains unchanged until the data-byte is received by the device.

If the device is programmed to be an output device and the DBA bit is set, it means that the data buffer is available for writing. When an output byte is written to the data register by the program, the bit is cleared. It remains cleared until the device has finished transmitting the data; it is set by the device to indicate that the transmission has completed. DBA is a status bit.

OV: Overrun (bit 3). This bit is set if the device is enabled for input and a byte is overwritten by a subsequent byte (i.e., it was not read quickly enough) or the device is enabled and the CPU writes bytes to the buffer before the device has time to transmit the byte. This is a status bit.

Reserved: These bits are reserved for future use or device-specific encodings (bits 4 through 7).

Data (I/O): These bits are for input or output, depending on how the device has been defined and subsequently enabled (bits 8 through 15).

Device ports are located in addresses 0x0000 through 0x000E. Each device is associated with a device number (ranging from 0 to 7).

2 Programming devices in X-Makina assembly language

As was shown in class on 14 June 2018, an X-Makina assembly language program can access a device. To simplify accessing the devices, locations should be reserved for the device's ports; for example, for devices 0 and 1:

```
        org  #$0    ; Device memory
DEV0    byte #0     ; Device 0 control/status register
        byte #0     ; Device 0 data register
DEV1    byte #0     ; Device 1 control/status register
        byte #0     ; Device 1 data register
```

In order to access the device, it is necessary to assign the address of the device to a register, which can then be used with one on the load-store instructions. For example, to access device 0, one could write (why this instruction?):

```
  movlz  DEV0,R0  ; R0 = address of DEV0
```

Register R0 now contains the address of device 0 ($0000).

Before the device can be used, it must be programmed to support either input (I/O = 1) or output (I/O = 0). Programming a device requires writing to the control/status register. In this example, the device is for input, therefore $02 (why?) must be written to the first byte (why?):

```
  mov.b  #$2,R1   ; R1 = 2
  st.b   R1,R0    ; Control/Status byte . IO = 1
```

The device is now programmed to support input.

When the device receives input, it will set the DBA (Data Byte Available) bit (DBA = 1). Prior to that point, the DBA bit is clear (DBA = 0). A polling loop can be employed to monitor the DBA bit ($4, bit pattern 0100, corresponds to the DBA bit's position in the control/status register):

```
InLoop
  ld.b   R0,R1    ; R1.byte = mem[DEV0] ; control/status register
  and.b  #$4,R1   ; R1 = R1 & DBA; PSW.Z=1 if DBA=0
  beq    InLoop   ; If result is zero (nothing received), repeat
```

When the DBA bit is set, data has been received, placed in the buffer by the device, and can be read by the program. The branch-if-equal instruction fails and control passes to the next record. The data byte can be read using a relative-addressing load (how and why?):

```
  ldr.b  R0,#1,R1 ; R1.byte = mem[Address of DEV0 + 1]
```

Accessing the output device requires that the device be programmed for output by clearing the I/O bit (I/O = 0). Assuming device 1 (location $0002) is to be used for output, one could write (why?):

```
  mov.b  #0,R1          ; R1 = 0
  str.b  R1,R0,#$2      ; mem[Address of DEV0 + 2] = 0
```

The device is now programmed to support output.[1]

[1] This zeros all bits in device 1's control/status port, which might not be a good idea as it means the current state of the device is lost (why?). The same thing occurred when programming device 0. However, since this is the

The status of the device determines if the output device is ready for output. In this case the DBA (Data Buffer Available) is inspected, if set (DBA = 1), a byte can be written to the output data register, otherwise the program must wait until the device is ready to transmit again, indicated by DBA being clear (DBA = 0). The DBA bit is usually not cleared immediately after a character is written to it because the output device may take time to process the byte before it is actually sent.

In this example, the polling loop is written as (why is load-relative used with R0 and a value of 2?):

```
OutLoop
  ldr.b  R0,#2,R2 ; R2.byte = mem[DEV0 + 2] ; control/status register
  and.b  #$4,R2   ; R2 = R2 & DBA
  beq    OutLoop  ; if result is zero (still xmitting), continue polling
```

If the branch-if-equal instruction fail, control passes to the next instruction. In this example, the character read previous is written to the output device:

```
  str.b  R1,R0,#3 ; mem[DEV0 + 3] = R1 (data read)
```

3 Emulating X-Makina's devices

The previous section showed how an X-Makina program could interact with two devices. Now the question is, how can the X-Makina emulator emulate devices? This question can be answered in two parts: first, where will the data come from and be written to? And second, what needs to be done inside the emulator to support this?

3.1 Data sources and sinks

Emulating devices is straightforward: input from input devices comes from a file and output from the CPU is written to an output file.

3.1.1 The input file

The input file is a series of records, each of which indicate three things, the time the data arrives from the device, the number of the device supplying the data (0 to 7), and the input data (such as a character). The records are sorted by time (i.e., time is increasing). For example,

```
10   3 A       <-- Time 10, device 3 receives an 'A'
27   4 X       <-- Time 27, device 3 receives an 'X'
85   3 b       <-- Time 85, device 3 receives an 'b'
106  2 3       <-- Time 106, device 2 receives an '3'
```

This implies that the CPU needs to maintain a system clock that increases and can be used to determine when the next character is to be written to the device specified in the input file. A random number generator producing random times, devices, and data are inadvisable because the sequence cannot be repeated, meaning that if errors are detected, they are not easily traced, whereas with the input file, specific conditions can be tested and, if an error is detected, can be repeated.

---

initialization sequence and should only be done once, prior to using the device for the first time, the device is probably in an undefined state, so it is considered acceptable.

The time between the arrival of data from a device is dictated by the time value in the input file. A rapid arrival rate from a device can be emulated by decreasing the time between the device's inputs. This can be used to test for overflow.

3.1.2 The output file

The output from the output devices needs to be recorded for testing and to demonstrate that the output devices are working as expected. This can be done by writing records to an output file, with each record containing the time the output occurred (from the CPU's system clock), the number of the device, and the output data. The output file can also be used to record other information, such as when a device processed its input data or whether an overflow occurred. Again, this can be useful to debugging purposes.

3.1.3 Device initialization

It is also necessary to indicate whether a device is used for input or output (it probably shouldn't hard-coded into the emulator – why not?). Output devices also needs the time required by the device to generate the output (i.e., the time between supplying the data and actually writing it to the output file).

This information can be placed as the first eight records in input file; for example:

```
1  0    <-- Record 0 - Dev 0 - Input (1) and processing time 0
0  100  <-- Record 1 - Dev 1 - Output (0) and processing time 100
0  250  <-- Record 2 - Dev 2 - Output (0) and processing time 250
...
1  0    <-- Record 7 - Dev 7 - Input (1) and processing time 0
```

Having the device's input/output characteristic specified this way makes more sense than having an X-Makina assembly language determine it, since most devices are hardwired as either input or output (or both).

3.2 Representing a device

The emulator must maintain information (i.e., state) of each device. For example, it needs to know whether the device is being used for input or output, whether data has been read from an input device, or how much time remains before the output device "finishes" its output.

This can be maintained in a structure; for example (this might be incomplete):

```
struct device
{
enum IO io : 1;          /* IO: INPUT or OUTPUT */
unsigned int dba : 1;    /* Input read? Output completed? */
unsigned int of : 1;     /* Overflow detected */
signed int proc_time;    /* Time to processes an output byte */
signed int time_left;    /* Time remaining to finish output */
unsigned char data;      /* Byte read or written */
};
```

Maintaining this information is useful, even if it is repeated in the device's memory, as it ensures that the X-Makina emulator knows the device characteristics, regardless what an assembly language writes to a device's register in memory.

Since there are eight devices, all eight can be represented in an array:

```
#define NUMDEV    8
struct device dev_array[NUMDEV];
```

The devices can initialized from the first eight records of the input file:

```
FOR DevNum = 0 TO NUMDEV-1
   READ dev_array[DevNum].io, dev_array[DevNum].proc_time
   dev_array[DevNum].dba = 0
   dev_array[DevNum].of = 0
   dev_array[DevNum].data = NUL
END FOR
```

3.3 Emulating devices

When the CPU is running, it will require a system clock that increments by a certain amount after each instruction is executed. This clock (or another) can be used to determine when the next character arrives or the time remaining in for an output to complete.

One question that needs answering is, how does the X-Makina emulator know when a device location has been read or written to by an X-Makina assembly language program?

The emulator can detect accesses to the device memory (i.e., primary memory locations 0x0000 through 0x000E) in the bus() function – if the address in the MAR is less than 0x0010, then a device location is being accessed. Dividing the address by 2 gives the device number. For example, if location 0x0001 is read (device 0's data register), it is necessary to check if the device is an input device (as this means the DBA bit can be cleared):

```
IF MAR < 0x0010 THEN
   DevNum = MAR / 2
   IF MAR IS ODD THEN
      IF CTRL = READ THEN
         IF dev_array[DevNum].io = INPUT THEN
            dev_array[DevNum].dba = 0
            memory[DevNum * 2].dba = 0
         ENDIF
         MBR = memory[MAR]
      ELSE /* WRITE */
         IF dev_array[DevNum].io = OUTPUT THEN
            dev_array[DevNum].dba = 0
            memory[MAR] = MBR
            dev_array[DevNum].time_left = dev_array[Dev_Num].prod_time
            dev_array[DevNum].data = MBR
         ENDIF
      ENDIF
   ENDIF
ELSE
   /* Normal memory access */
ENDIF
```

As part of the instruction cycle (preferably after the instruction has been executed), it is also necessary to determine if a device has supplied a character. This means comparing the system clock with the time of the most recently read input file record (stored in InTime, InDevice, and InData):

```
IF SysClock >= InTime THEN
   IF dev_array[InDevice].io = INPUT THEN
```

```
    IF dev_array[InDevice].dba = 0 THEN
      dev_array[InDevice].dba = 1
    ELSE /* Last byte not read - overrun error */
      dev_array[InDevice].ov = 1
      memory[InDevice * 2].ov = 1
      memory[InDevice * 2 + 1] = InData
    ENDIF
  READ InTime, InDevice, InData /* Next record in Input file */
ENDIF
```

Remember, there might be more than one device sending data to the machine at the same time.

It is also necessary to determine if there is pending output for any of the output devices:

```
IF dev_array[output_device].time_left THEN
  dev_array[output_device].time_left = dev_array[output_device].time_left -
    decrement
  IF dev_array[output_device].time_left <= 0 THEN
    /* Write to output file: */
    WRITE SysClock, output_device, dev_array[output_device].data
    dev_array[output_device].dba = 1
    memory[output_device * 2].dba = 1
  ENDIF
ENDIF
```