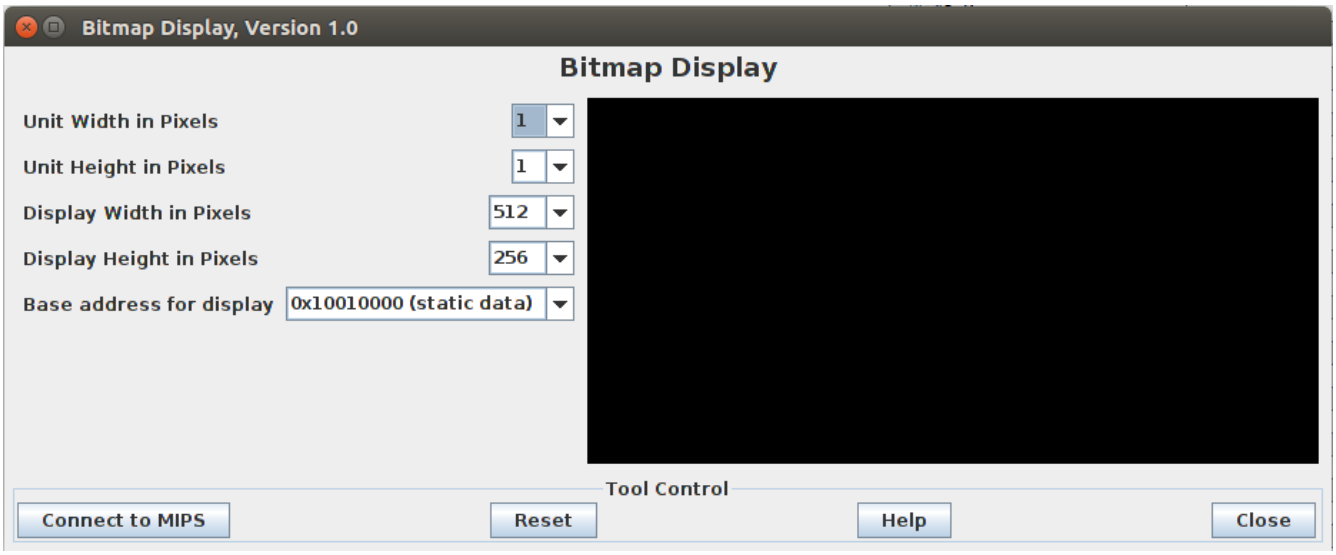
MARS MMIO Bitmap and Keyboard Tutorial

Dr. Karen Mazidi
https://github.com/kjmazidi/CS3340

The MARS bitmap uses a method of input/output called memory-mapped I/O, or MMIO. The top left pixel of the display, coordinates (0, 0), is mapped to address 0x10010000, which is the start of the data segment. The pixel size and screen size both have default values, discussed below. The color of a pixel is a 24-bit RGB value stored as a word. The default is black 0x00000000 so only the non-black colors need to be stored. Adding color to a pixel location is simply storing the hex for that color in the location, and returning it to black is done by storing 0 in that location. See more color codes at: https://www.rapidtables.com/web/color/RGB_Color.html

To open the MARS bitmap display, assemble a program, then go to the Tools menu, then Bitmap Display. You will see the default settings:



Notice that you can change the dimensions of the display as well as the pixels. You can change the base address for the data as well.

With the default settings you would have 512 x 256 pixels, which enables fine-grained images to be created.

A sample program

The following program shows the basics of drawing a pixel to the display. No .data section is needed for this program. There are several constants defined at the top of the program with the .eqv directive. These work just like #define in C. The assembler will replace the equivalent text with the number while assembling the program. The WIDTH and HEIGHT are 64, assuming that the pixel dimensions are changed from the default 1x1 to 4x4, and the display dimensions are set to 256x256.

Since the display is 256 pixels wide and each 'pixel' is 4 bits, this means that there are 64 4x4 pixels in each row, and there are 64 rows. The code also has .eqv values for colors red, green and blue.

Here is the code for this setup:

```
1  # Bitmap Demo Program
2  # no .data section initialized
3  # Instructions:
4  #         set pixel dim to 4x4
5  #         set display dim to 256x256
6  # Connect to MIPS and run
7
8  # set up some constants
9  # width of screen in pixels
10 # 256 / 4 = 64
11 .eqv WIDTH 64
12 # height of screen in pixels
13 .eqv HEIGHT 64
14 # memory address of pixel (0, 0)
15 .eqv MEM 0x10010000
16 # colors
17 .eqv    RED     0x00FF0000
18 .eqv    GREEN   0x0000FF00
19 .eqv    BLUE    0x000000FF
20
```

The following code is a subroutine to draw a pixel at ($a0, $a1) in color $a2:

```
41
42 ################################################
43 # subroutine to draw a pixel
44 # $a0 = X
45 # $a1 = Y
46 # $a2 = color
47 draw_pixel:
48         # s1 = address = MEM + 4*(x + y*width)
49         mul     $s1, $a1, WIDTH   # y * WIDTH
50         add     $s1, $s1, $a0     # add X
51         mul     $s1, $s1, 4       # multiply by 4 to get word offset
52         add     $s1, $s1, MEM     # add to base address
53         sw      $a2, 0($s1)       # store color at memory location
54         jr      $ra
55
```

In this program WIDTH = 64, the number of 4x4 pixels per row.

The memory locations are stored in row order. So a display location at (2, 3) would correspond to a memory offset of 3 * WIDTH + 2. This value would need to be multiplied by 4 to get the word offset, then added to the base address for the memory map.

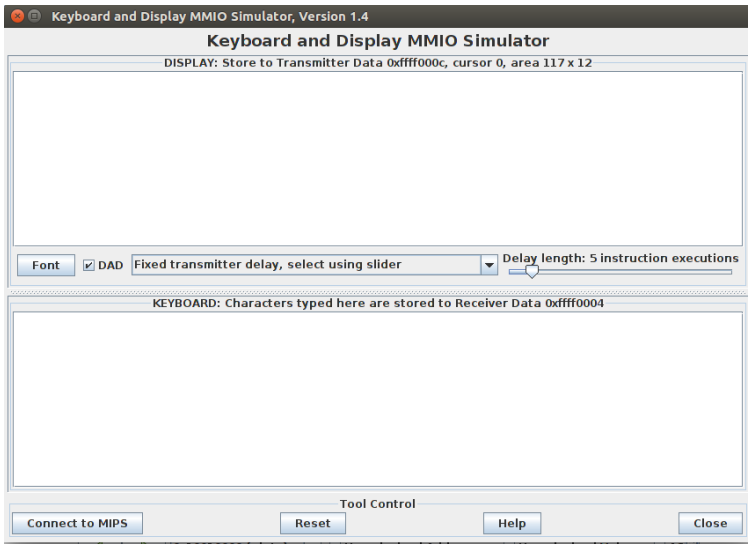
The following code calls the draw_pixel subroutine 3 times. The first pixel calculates the center of the screen (x, y) as (WIDTH/2, HEIGHT/2). The color of the first pixel is red. To draw the second pixel, 1 is added to $a0, the X value, and the color in $a2 is changed to GREEN. To draw the third pixel, 1 is added to $a0 and the color in $a2 is changed to BLUE.

```
20
21  .text
22  main:
23          # draw a red  pixel in the center of the screen
24          addi    $a0, $0, WIDTH    # a0 = X = WIDTH/2
25          sra     $a0, $a0, 1
26          addi    $a1, $0, HEIGHT   # a1 = Y = HEIGHT/2
27          sra     $a1, $a1, 1
28          addi    $a2, $0, RED  # a2 = red (ox00RRGGBB)
29          jal     draw_pixel
30          # now draw a green one to the right
31          addi    $a0, $a0, 1
32          addi    $a2, $0, GREEN
33          jal     draw_pixel
34          # now draw a blue one to the right
35          addi    $a0, $a0, 1
36          addi    $a2, $0, BLUE
37          jal     draw_pixel
38
39  exit:   li      $v0, 10
40          syscall
41
```

In this program, the MMIO data was stored at the start of the .data section, 0x10010000. This makes it difficult to add other memory locations that the program may need. A more common approach is to set up program data in the .data section, then use the area starting at $gp for the MMIO data. This is done by selecting $gp in the drop-down menu for base address. Also, the code to compute the pixel address would be changed to add $gp instead of MEM. The second example program in this tutorial uses $gp for the MMIO data.

The Keyboard Simulator

MARS also has a keyboard simulator, available in the Tools menu, shown below:



The top pane is an output area. The bottom pane accepts keyboard input and echoes it back. The memory mapped I/O for the keyboard uses 64KB of memory from 0xffff0000 to 0xffffffff. There are two special locations we can think of like 'registers' within this memory:

- A control register at 0xffff0008 – bit 0 is the ready bit; bit 1 is an interrupt-enable bit
- A data register at 0xffff000c – only the lower byte is used; when a character is written there, the printer will display that character

In a real computer system, the data in the data register could be changed faster than the display can keep up with. This would result in characters being ignored in the output. To solve this, the display/printer can control the ready bit by changing it to 1 to indicate that a character can be written, or 0, which indicates that a character should not be written.

Polling

A simple way to manage this is to keep a loop going that waits for the display/printer to be ready. The code might look something like this, assuming that the character to be written is in $s0:

```
poll: lw    $t0, CTRLREG     # load the control reg into $t0
      andi  $t0, $t0, 1      # if readybit=1, then $t0 = 1, else $t0 = 0
      beq   $t0, $0, poll    # keep waiting if not ready
      sb    $s0, DISPREG     # write the char to the display register
```

The problem with this is that you are keeping the CPU busy in this loop and it can't do anything else. Early computers worked this way with the keyboard, but then the CPU's weren't that fast. The lag could be somewhat overcome with keeping a buffer of keyboard input, which would be processed every second or so.

Interrupt

A better way is to use interrupts for keyboard input. When the interrupt bit in the control register (bit 1) changes from 0 to 1, this triggers the interrupt. Control is passed from whatever the CPU was doing, to an interrupt-service-routine. When the interrupt-service-routine is finished handling the interrupt, control pases back to whatever the CPU was doing before it was interrupted. MIPS interrupts are handled by co-processor 0.

Back to the MARS keyboard simulator above. In the bottom pane, the ready bit is normally off. When a key is typed in this pane, the character is placed in the data register and the ready bit is turned on. The character can be loaded from the data register as soon as the ready bit becomes 1. When the data is read from the data register, the ready bit is reset to 0. If the interrupt-enable bit is on when the ready bit goes on, an interrupt will be triggered. The registers used for this pane are 0xffff0004 for data and 0xffff0000 for control.

Sample Program 2

This program uses both the bitmap display and the keyboard simulator. The program draws a large red pixel roughly in the center of the screen. The pixel will 'move' when the user presses w, a, s, or d. If the user presses space, the program terminates.

For this program, the bitmap settings are changed so that each pixel is 8x8. The screen is still 256x256. However, since each pixel takes 8 bits, the width and height in pixels is 32. Here are the changes to the .eqv section of code:

```
# set up some constants
# width of screen in pixels
# 256 / 8 = 32
.eqv WIDTH 32
# height of screen in pixels
.eqv HEIGHT 32
# colors
.eqv    RED     0x00FF0000
```

The program has been modified from the previous program to use $gp as the offset instead of the data segment. This would allow use of the .data section for things like error message storage, instructions, etc. However, this program still just has a .text section.

The subroutine to draw the new pixel is very similar to the earlier draw pixel, but $gp is used.

```
################################################
# subroutine to draw a pixel
# $a0 = X
# $a1 = Y
# $a2 = color
draw_pixel:
        # s1 = address = $gp + 4*(x + y*width)
        mul     $t9, $a1, WIDTH   # y * WIDTH
        add     $t9, $t9, $a0     # add X
        mul     $t9, $t9, 4       # multiply by 4 to get word offset
        add     $t9, $t9, $gp     # add to base address
        sw      $a2, ($t9)        # store color at memory location
        jr      $ra
```

The main code has an infinite loop of drawing the pixel while waiting for keyboard input. The MMIO keyboard is connected with the default settings prior to running the program. The input shows in the bottom pane only, the top pane is not used in this program. The first section of code below draws the big red pixel roughly in the center of the screen. The address 0xffff0000 is loaded into $t0. If this is zero, then no input is available. If it is not zero, the input is in address 0xffff0004. This is loaded into register $s1.

The wasd keys are used for up, left, down, and right. The space key means exit the program. The code below checks that one of these 5 has been pressed, otherwise it loops, ignoring any invalid input.

```
.text
main:
        # set up starting position
        addi    $a0, $0, WIDTH    # a0 = X = WIDTH/2
        sra     $a0, $a0, 1
        addi    $a1, $0, HEIGHT   # a1 = Y = HEIGHT/2
        sra     $a1, $a1, 1
        addi    $a2, $0, RED  # a2 = red (ox00RRGGBB)

loop:   # draw a red  pixel
        jal     draw_pixel

        # check for input
        lw $t0, 0xffff0000  #t1 holds if input available
        beq $t0, 0, loop   #If no input, keep displaying

        # process input
        lw      $s1, 0xffff0004
        beq     $s1, 32, exit   # input space
        beq     $s1, 119, up    # input w
        beq     $s1, 115, down  # input s
        beq     $s1, 97, left   # input a
        beq     $s1, 100, right # input d
        # invalid input, ignore
        j       loop
```

The series of beq statements check for valid input and jump to the appropriate code, shown below:

```
        # process valid input

up:     li      $a2, 0          # black out the pixel
        jal     draw_pixel
        addi    $a1, $a1, -1
        addi    $a2, $0, RED
        jal     draw_pixel
        j       loop

down:   li      $a2, 0          # black out the pixel
        jal     draw_pixel
        addi    $a1, $a1, 1
        addi    $a2, $0, RED
        jal     draw_pixel
        j       loop

left:   li      $a2, 0          # black out the pixel
        jal     draw_pixel
        addi    $a0, $a0, -1
        addi    $a2, $0, RED
        jal     draw_pixel
        j       loop

right:  li      $a2, 0          # black out the pixel
        jal     draw_pixel
        addi    $a0, $a0, 1
        addi    $a2, $0, RED
        jal     draw_pixel
        j       loop

exit:   li      $v0, 10
        syscall
```

Each section of code up/down/left/right first blacks out the current pixel by setting the color to 0 and calling the draw_pixel subroutine. Then the x ($a0) or y ($a1) values are updated accordingly, the pixel color is changed back to red, and the pixel is drawn in the new location. After drawing the new pixel, the code continues looping, drawing the pixel while waiting for keyboard input. This is keeping the CPU busy drawing the same pixel most of the time, which is wasteful of CPU resources. This approach is the polling method, which is less efficient in terms of CPU time than an interrupt method.

This tutorial has shown how to draw to the bitmap display and use the keyboard to control action. A popular use of these techniques is to create a game like Snake; many examples can be found on the web.