Tutorial 5: Old McMicrocontroller Had a Port, G-P-I-P-I-O

Prof. John McLeod

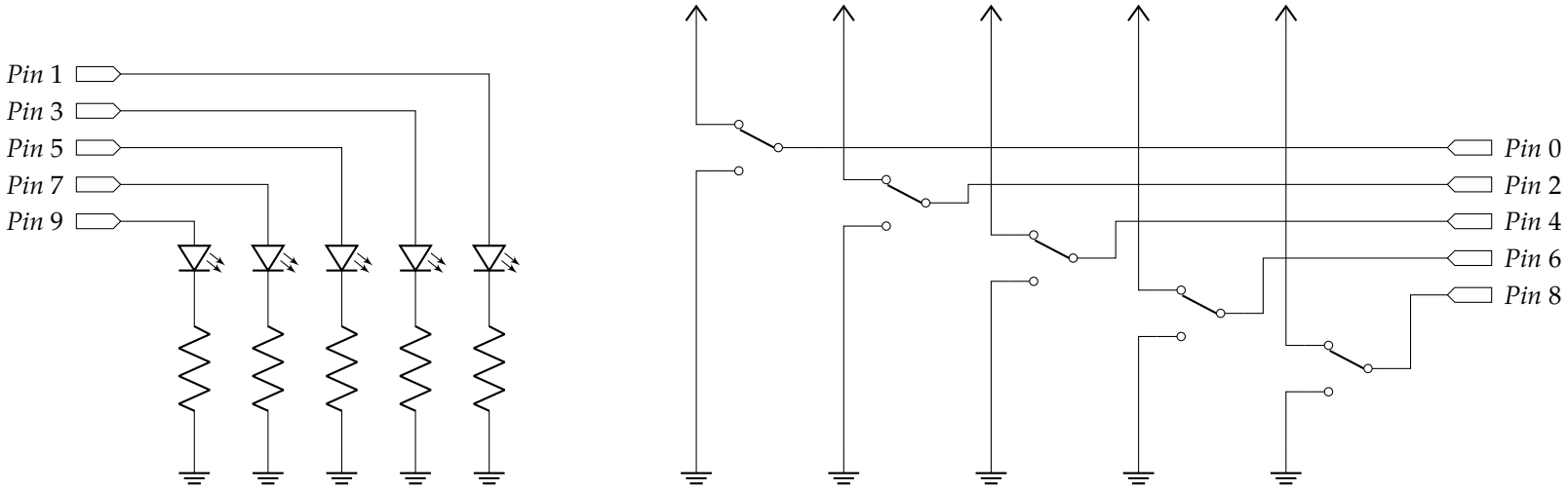
ECE3375, 2021 02 25

The DE1-SoC simulator has two 32-pin GPIO ports at `0xff200060` and `0xff200070`. On a real board, we would have to connect a parallel port jack to the board and connect to some other external peripheral in order to interact with these pins — but in the simulator the status of the pin as input or output, and whether or not the pin as a 0 or 1 is visually represented, and input-configure pins can be toggled by clicking on them. What a wonderful pedagogical tool!

Problem: We want to use the lowest 10 pins on the first GPIO port to connect to the built-in LED and switch bank peripherals. Specifically:

- All odd-numbered pins should be configured as input, and that input should be displayed on the corresponding odd-numbered built-in LEDs.

- All even-numbered pins should be configured as output, and that output should write the state of the corresponding even-numbered built-in switches.



Basically, this project is a fancy way of making the microcontroller completely unnecessary — as the probelm is stated, wiring the LEDs directly to the input pins and the switches directly to the output pins would have the same effect. Obviously this is the kind of project that management will love.

Solution: First, we need to set the direction of the ports on the control register. In an ARM system, the control register is at the base address of the GPIO port, offset by `0x04`. As discussed in the lecture, the following structure is useful:

```
#define GPIO_A_BASE     0xFF200060

typedef struct _gpio
{
  int data;
  int control;
} gpio;

volatile gpio* const port = (gpio*) GPIO_A_BASE;
```

To set the direction of the ports, we should use a bit mask. We only need to set the bottom 10 pins, it is best to leave the remaining pins unchanged in case they are used by some other part of the program. Recall a control bit set to 0 acts as an input, a control bit set to 1 acts as an output.

- To set pins 1, 3, 5, 7, and 9 to zero, we should prepare a mask of all 1s *except* at bits 1, 3, 5, 7, and 9. We then use a bitwise `AND` to apply this mask.
- To set pins 0, 2, 4, 6, and 8 to one, we should prepare a mask of all 0s *except* at bits 0, 2, 4, 6, and 8. We then use a bitwise `OR` to apply this mask.
- Since each operation leaves all remaining bits unaffected, we

can apply these two masks sequentially and at the end we will have the appropriate pins set to input and output.

- It is worth recognizing that if a continuous set of pins are all to be set as input or output, the *only* difference between the output and input masks is whether the remaining bits (in our case, bits 10 to 31) are ones or zeros.

There are a few different ways of creating the bit masks. Probably the most direct way is to just calculate the appropriate decimal number that corresponds to the desired bit sequence:

$$\text{Input mask:} \quad x_{\text{in}} = \left(2^{32} - 1\right) - \sum_{n \in S_{\text{in}}} 2^n,$$

$$\text{Output mask:} \quad x_{\text{out}} = \sum_{n \in S_{\text{out}}} 2^n,$$

$$\text{where here:} \quad S_{\text{in}} = \{1, 3, 5, 7, 9\}\,, \quad \text{and} \quad S_{\text{out}} = \{0, 2, 4, 6, 8\}\,.$$

For this collection of pins, $x_{\text{in}} = 4\,294\,966\,614$ and $x_{\text{out}} = 341$.

Another way is to use a few fancy tricks with bitwise logic, as there is a pattern to the bits set for input and output. Really it is a matter of personal preference, but I usually find a bitwise logic expression is easier to read — at least in terms of understanding which bits are zeros and which are ones — than some arbitrary-looking decimal number.

```
// output mask is mostly zeros with selected 1s
int output_mask = 1 + (1<<2) + (1<<4) + (1<<6)
                  + (1<<8);
// for this situation, input mask is complement
// of output mask shifted left by one bit
// and 0th bit set to 1 again
int input_mask = ~(output_mask << 1) + 1;

// apply the masks
port->control &=input_mask;
port->control |=output_mask;
```

Now it is just the relatively simple matter of reading from, or writing to, the port, and interacting with the LEDs and the switches. The following variables will be used:

```
#define LED_BASE      0xFF200000
#define SWITCH_BASE   0xFF200040

volatile int* const led = (int*)LED_BASE;
volatile int* const switches = (int*)SWITCH_BASE;
```

As the DE1-SoC board has only 10 LEDs, is it as simple as reading from the port and write directly to the LED bank?

```
//read from port and write to
// LEDs in one step
*led = port->data;
```

This will work initially, but what happens if there was output previously written to the port?

```
// read from switches and
// write to port in one step
port->data = *switches;

//read from port and write to
// LEDs in one step
*led = port->data;
```

Now the LEDs light up according to the pins that were previously *written* to high on the last output! Apparently, we are not done with masks.

- To isolate data from only the input pins, we need to prepare a mask of all zeros *except* at pins 1, 3, 5, 7, and 9. We then use a bitwise `AND` to apply this mask.
- The mask required is just the complement of the original input mask.

```
//read from port and write to
// LEDs in one step
*led = (port->data & ~input_mask);
```

It is less necessary to mask the switch inputs, at least in this example and when using the simulator, but it might be a good idea in general.

- If *all* of the lowest 10 GPIO pins were set as output, but only *some* of them were supposed to output from the switches, it would be necessary to mask the switch input to avoid accidentally setting other pins.
- Depending on how the microcontroller is internally wired, attempting to write an *output* to an *input* pin that is different than the current input value may cause problems.
- Furthermore, even if the GPIO port is internally wired such that attempting to write to an input pin does not cause any problems, if the port has other output pins set *outside* the lowest 10, simply overwriting the port data with the switches may erroneously change those other outputs.

To address the first issue, we should mask the switch data to remove the input from all unwanted switches. To address the second issue, we should apply a mask to the current state of the GPIO port.

- To isolate data only from the appropriate input switches, we need to prepare a mask of all zeros *except* at switches 0, 2, 4, 6, and 8. We then use a bitwise `AND` to apply this mask.
- This is the same as the original output mask.
- To retain data from all GPIO pins except the appropriate output pins, we need to prepare a mask of all ones *except* at pins 0, 2, 4, 6, and 8. We then use a bitwise `AND` to apply this mask.
- This is the same as the complement of the original output mask.

- To only modify the data at the appropriate output pins, we can combine both masked results with a bitwise `OR`.

```
//read from switches, mask, and
// write to GPIO port in one step
port->data = (port->data & ~output_mask)
             | (*switches & output_mask);
```

The following code combines all of these steps, and operates in an endless loop. Note:

- In this code I added an extra line to set a bunch of arbitrary pins on the GPIO port to input/output.
- This is to help demonstrate that rest of the program *only* affects the bottom 10 GPIO pins.
- On the physical DE1-SoC board, there are only 10 LEDs and 10 switches. However in the simulator each can be expanded to 32 — just click on the small "down arrow" on the peripheral's window to see that option.
- If you expand the LEDs and switches to 32, running this program should still only read/write from the bottom 10: LEDs 10 to 31 will never light up, and the state of switches 10 to 31 will be ignored.
- This again helps demonstrate that the bit masks are working correctly.

```
#define LED_BASE        0xFF200000
#define GPIO_A_BASE     0xFF200060
#define SWITCH_BASE     0xFF200040

typedef struct _gpio
{
  int data;
  int control;
} gpio;

volatile gpio* const port = (gpio*) GPIO_A_BASE;
volatile int* const led = (int*)LED_BASE;
volatile int* const switches = (int*)SWITCH_BASE;

void main()
{
  // output mask is mostly zeros
  // 1s at bits 0,2,4,6,8
  int output_mask = 1 + (1<<2) + (1<<4) + (1<<6)
                    + (1<<8);
  // here input mask is complement
  // of shifted output mask
  int input_mask = ~(output_mask << 1);

  ////////////////////////////////////
  // this isn't part of the design problem
  //
```

```
  // this sets a bunch of pins to input and
  // output, just to show that our masks work
  // correctly by only using bottom 10 pins
  port->control = 1885922191;
  ////////////////////////////////////////

  // apply masks to set pin direction
  port->control &= input_mask;
  port->control |= output_mask;

  // endless loop
  while(1)
  {
    // read from port, mask, and write to
    // LEDs in one step
    *led = ( port->data & ~input_mask );

    // read from switches, mask, also
    // mask current port state, and
    // write to port in one step
    port->data = ( port->data & ~output_mask )
                 | ( *switches & output_mask );
  }
}
```