Lab 2: Intro to AVR Toolchain

UCR EE/CS 120B

Before you start developing, you need to create a Github account for yourself if you do not already have one. The files that you create on the lab machines are only temporary. You will need to push the changes to github to save your work.

Pre-lab

Read the lab manual and write the main C code for exercises 1 and 2 targeting the atmega1284.

Tools

AVR-GCC

The GCC compiler that specifically targets the AVR 8-bit RISC microcontrollers including the Atmega1284 used in this course. AVR-GCC works similarly to GCC with one key difference, you need to specify the target

- `-mmcu=<device>` command line option will specify the target device to generate code for. The ATmega1284 is `atmega1284`

The command to compile the source code file(s) into an “Executable and Linkable Format” (ELF) file is

```
$avr-gcc -Wall -g -Og -mmcu=atmega1284 <source file(s)>
```

The `-g` flag will produce debugging information to be used by AVR-GDB and `-Og` will ensure that compiler passes that produce useful debugging information will actually run.

AVR-GDB

The Gnu Debugger tool can be useful for debugging your microcontroller on the host system similar to GDB. You are able to set breakpoints (`break [b]`), step through the code, continue to execute until the next break point (`continue[c]`), skip `N` break points (`continue N`), inspect results (`display <var>`), and change input (`set <var>=<value>`). For more information on working with AVR-GDB, follow this tutorial.

SimAVR

We will be using SimAVR, “a lean, mean and hackable AVR simulator for linux & OSX”. If you would like to setup your own development environment you can follow the installation instructions on the github page for SimAVR. If you are working on Windows we recommend using Atmel Studio.

SimAVR will simulate your `.elf` object file, similar to a `.o` file, running on the Atmega1284 so you can “verify” that everything is working correctly before programming. There is no guarantee that simulated results will be 100% accurate, final testing must always be done on the target device. Additionally, it allows you to create a Value Change Dump (VCD) file that can visualize the changes on the ports and pins using a waveform generator like *gtkwave*.

Creating and compiling a new project

1. Create a new example project
   ```
   $ /usr/csshare/pkgs/cs120b-avrtools/createProject.sh
   Project name: Lab2_introToAVR
   Partners name [none]: Jo Smith
   Microcontroller [atmega1284]:
   Clock Frequency [8000000]:
   ```
   Note: The value in square brackets `[ ]` is the default if you don't enter anything.
2. This will generate the following directory structure
   ```
    Lab2_introToAVR
   ├── build
   │   ├── bin
   │   ├── objects
   │   └── results
   ├── header
   │   └── simAVRHeader.h
   ├── Makefile
   ├── source
   │   └── main.c
   ├── test
   │   ├── commands.gdb
   │   ├── initDebugger.gdb
   │   └── tests.gdb
   └── turnin
   ```
3. Now, we will write a simple program (shown below) that sets port B's 8 pins to 0000 1111

```c
#include <avr/io.h>

int main(void){
	DDRB = 0xFF; // Configure port B's 8 pins as outputs
	PORTB = 0x00; // Initialize PORTB output to 0's
	while(1){
		PORTB = 0x0F; // Writes port B's 8 pins with 00001111
	}
	return 1;
}
```

The template `main.c` provided has an additional header "`simAVRHeader.h`" that we will use in the next section.

4. Compile your program. *You should be in the topmost directory of your project.* Don't forget to specify the target device!
   ```
   $ avr-gcc -mmcu=atmega1284 -Wall -o build/objects/main.elf source/main.c
   ```
5. You should see a `main.elf` file in your `build/objects` folder (`ls build/objects`).
   Congratulations you built your first "embedded" program!

Simulating your new project

To simulate your program using SimAVR, we need to add a few data structures. We have added these into the `./header/simAVRHeader.h` file which is conditionally included in your `main.c`

```
#ifndef F_CPU
#define F_CPU 8000000 // The atmega1284 has an 8MHz clock
#endif
#include <avr/sleep.h>
#include "include/simavr/avr/avr_mcu_section.h"
AVR_MCU(F_CPU,"atmega1284");
AVR_MCU_VCD_FILE("build/results/Lab2_introToAVR_trace.vcd",1000);

const struct avr_mmcu_vcd_trace_t _mytrace[] _MMCU_ = {
	{ AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, } ,
};
```

Where

- `F_CPU` will define the clock frequency (8MHz in the ATmega1284)
- `AVR_MCU(...)` lets SimAVR know that we are simulating on the ATmega1284 with an `F_CPU` clock
- `AVR_MCU_VCD_FILE(...)` allows us to name the generated trace file
- `avr_mmcu_vcd_trace_t` is a struct used by SimAVR to know which signals to trace and generate a wave form. You can set the name of the traced symbol using `AVR_MCU_VCD_SYMBOL(...)`, set the signal using `.what=(void*)&<signal>`, and even set a mask if you only want one or two bits (we'll see an example of this later).

Now, we need to simulate the program we've written.

1. Rebuild the project. Run this command in the topmost directory of your project.
   ```
   $ avr-gcc -mmcu=atmega1284 -I/usr/csshare/pkgs/simavr/ -Iheader/
   -D_SIMULATE_ -Wall -Wl,--undefined=_mmcu,--section-start=.mmcu=910000 -o
   build/objects/main.elf source/main.c
   ```
   We need to include all of the additional header files (`-I`) and define `_SIMULATE_` to include the `simAVRHeader.h`. Yes, this is tedious, but don't worry, we have a `Makefile` to help.
2. Run the simulator with the `-m <device>` and `-f <clock frequency>` flags (and yes, the multiple `-v` flags are not a typo, they change the verbosity level, put as many or as few as you would like for different levels of output).
   ```
   $ simavr -v -v -v -m atmega1284 -f 8000000 build/objects/main.elf
   Loaded 206 .text at address 0x0
   Loaded 0 .data
   Creating VCD trace file 'build/results/Lab2_introToAVR_trace.vcd'
   _avr_vcd_notify FIFO Overload, flushing!
   ...
   ```
3. The simulator will run forever since there is a `while(1)` loop in our program, exit the simulator using CTRL+C.
4. A file `Lab2_introToAVR_trace.vcd` should have been generated in the `build/results` folder (`ls build/results`). Note: the filename depends on what you named your project). Open it using gtkwave.
   ```
   $ gtkwave build/results/Lab2_introToAVR_trace.vcd
   ```
   For more information on working with gtkwave follow this tutorial.

5. Congratulations! You have now simulated your first “embedded” program! You will want to remove the `.vcd` file before continuing (`rm build/results/Lab2_introToAVR_trace.vcd`). Since we have a `while(1)`, the `.vcd` file gets quite large and takes up space on the lab machine.

Reading from input pins

The ATmega1284 has four 8-bit ports named A, B, C, and D, each port being 8 physical pins on the chip. Each port has

- a corresponding 8-bit Data Direction Register (DDR) `DDRA`, `DDRB`, `DDRC`, and `DDRD` that configure each port's pins to either an input (`0`) or an output (`1`).
  - Thus, for example "`DDRB = 0xFF;`" configures all 8 pins of port B to be outputs;
  - “`DDRB = 0xF0;`” configures the high nibble to output, and the lower nibble to input.
- Each port also has a corresponding 8-bit output register `PORTA`, `PORTB`, `PORTC`, and `PORTD` for writing to the port's physical pins (write only!) and
- a corresponding 8-bit Port Input (PIN) register `PINA`, `PINB`, `PINC`, `PIND` for reading from the port’s physical pins (read only!).

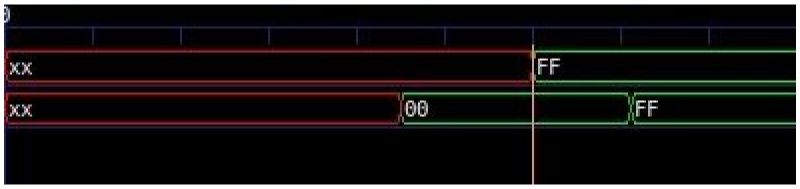
1. Modify the program into the following program:

```
#include <avr/io.h>
#ifdef _SIMULATE_
#include “simAVRHeader.h”
#endif

int main(void) {
        DDRA = 0x00; PORTA = 0xFF; // Configure port A's 8 pins as inputs
        DDRB = 0xFF; PORTB = 0x00; // Configure port B's 8 pins as outputs
                                   // Initialize output on PORTB to 0x00
        unsigned char temp_input = 0x00;
        while(1) {
                temp_input = PINA;
                PORTB = temp_input; // Writes port B's 8 pins with the values
                                    // on port A's 8 pins
        }
        return 1;
}
```

And add the following line to the `_MMCU_` structure in the `header/simAVRHeader.h` file
`{ AVR_MCU_VCD_SYMBOL(“PINA”), .what = (void*)&PINA, } ,`

2. Compile the program
`$ avr-gcc -mmcu=atmega1284 -I/usr/csshare/pkgs/simavr/ -Iheader/ -D_SIMULATE_ -Wall -Wl,--undefined=_mmcu,--section-start=.mmcu=910000 -o build/objects/main.elf source/main.c`
3. Simulate the program
`$ sim_avr -v -v -v -v -m atmega1284 -f 8000000 build/objects/main.elf`
4. Exit the simulator (CTRL+C) and visualize the output
`$ gtkwave build/results/Lab2_introToAVR_trace.vcd`



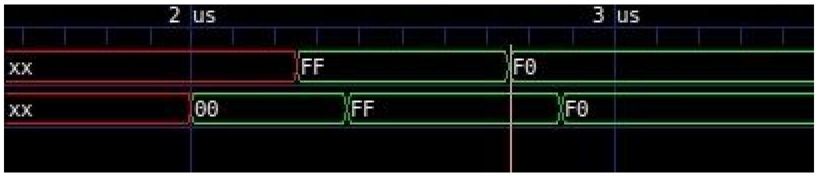
As you can see, `PINA` is initialized to `FF` (`PORTA = 0xFF`) and never changes (there is no input), and so

`PORTB` starts undefined (`xx`), changes to `00` on initialization and then becomes `FF` in the loop. We need to connect the simulator to AVR-GDB to change the pins and see how our program reacts.

1. Compile the program with the `-g` and `-Og` flags (debugging information and debugging optimizations)
   ```
   $   avr-gcc   -mmcu=atmega1284   -I/usr/csshare/pkgs/simavr/   -Iheader/
   -D_SIMULATE_ -Wall -g -Og -Wl,--undefined=_mmcu,--section-start=.mmcu=910000
   -o build/objects/main.elf source/main.c
   ```
2. Simulate the program again, this time pass in the `-g` flag for the interactive debugger.
   ```
   $ simavr -v -v -v -v -g -m atmega1284 -f 8000000 build/objects/main.elf
   Loaded 184 .text at address 0x0
   Loaded 0 .data
   Creating VCD trace file 'Lab2_introToAVR_trace.vcd'
   avr_gdb_init listening on port 1234
   ```
3. In another terminal (in the same directory), start the debugger with the commands file
   ```
   $ avr-gdb -x test/commands.gdb
   ```
4. Load the .elf file into the debugger so it knows the debug information
   ```
   (gdb) file build/objects/main.elf
   Reading symbols from build/objects/main.elf...done.
   ```
5. Add the remote target for the simulator and load
   ```
   (gdb) target remote :1234
   Remote debugging using :1234
   0x00000000 in __vectors ()
   ```
6. Add a break point at the top of the `while(1)`. Your line number (:#) may differ
   ```
   (gdb) break main.c:22
   Breakpoint 1 at 0xae: file main.c, line 22
   ```
7. Set the debugger to display the values we're concerned with:
   ```
   (gdb) commands
   Type commands for breakpoint(s) 1, one per line.
   End with a line saying just "end".
   >silent
   >printPINA
   >printPORTB
   >end
   (gdb)c
   Continuing.

   Breakpoint 1, main() at source/main.c:14
   14         temp_input = PINA;
   PINA addr0x800020:    0xff
   PORTB addr0x800025:   0x00
   ```
   Note: The "`Breakpoint 1` …" output won't show up if you made the `commands silent` above.
8. Now, in gdb, change the value `PINA` to `0xF0` to simulate input changing and continue several times. **Note**: in embedded systems it is typically a good time to run the loop several times to make sure the results are stable before inspecting them.
   ```
   (gdb) setPINA 0xF0
   (gdb) continue 5
   Will ignore next 4 crossings of breakpoint 1. Continuing.

   Breakpoint 1, main () at main.c:14
   14         temp_input = PINA;
   PINA addr0x800020:    0xf0
   PORTB addr0x800025:   0xf0
   ```
9. Quit the debugger (`quit`) AND exit the simulator (CTRL+c) and open the trace file



10. Congratulations, you have now simulated your first embedded program with input!

<u>Always strive to read from input (PINx) and write to output (PORTx).</u>
<u>Mixing these up may cause odd behavior.</u>

Accessing individual pins (bit masking)

1. Modify the program into the following program:

```
#include <avr/io.h>

int main(void) {
        DDRA = 0x00; PORTA = 0xFF; // Configure port A's 8 pins as inputs
        DDRB = 0xFF; PORTB = 0x00; // Configure port B's 8 pins as outputs, initialize to 0s
        unsigned char tmpB = 0x00; // Temporary variable to hold the value of B
        unsigned char tmpA = 0x00; // Temporary variable to hold the value of A
        while(1) {
                // 1) Read input
                tmpA = PINA & 0x01;
                // 2) Perform computation
                // if PA0 is 1, set PB1PB0 = 01, else = 10
                if (tmpA == 0x01) { // True if PA0 is 1
                        tmpB = (tmpB & 0xFC) | 0x01; // Sets tmpB to bbbbbb01
                                                     // (clear rightmost 2 bits, then set to 01)
                } else {
                        tmpB = (tmpB & 0xFC) | 0x02; // Sets tmpB to bbbbbb10
                                                     // (clear rightmost 2 bits, then set to 10)
                }
                // 3) Write output
                PORTB = tmpB;
        }
        return 0;
}
```

And add the following line to the `_MMCU_` structure in the `simAVRHeader.h` file
`{ AVR_MCU_VCD_SYMBOL("PINA0"), .mask = 1 << 0, .what = (void*)&PINA, } ,`

NOTE: Use of the notation "PA0", "PB1", and "PB0" in the comments to refer to port A's bit 0, port B's bit 1, and port B's bit 0, respectively. We will use such notation extensively in comments and lab assignment text, but realize that those are NOT recognized identifiers by the AVR C compiler.
NOTE: Use of a temporary variable tmpB instead of reading from PORTB.

2. Build and run the program as before.

The code shows a common method for reading one pin of a port: "`PINA & 0x01`" to read PA0 (as another example, "`PINA & 0x08`" would read PA3, resulting in 0x00 if PA3 was 0 or 0x08 if PA3 was 1). The code also shows a common method for writing to a particular bit (or bits) of a variable: "`tmpB = (tmpB & 0xFC) | 0x01`" first clears bit 1 and bit 0, then writes "01" to those bits. Note that the selected masks preserve the other bits of tmpB, which is then written out to port B.

Writing test cases

Using AVR-GDB manually and tracing through is great for debugging if there is something wrong with your code, but ideally, you should be able to write tests to verify the functionality. GDB allows us to write scripts to feed input to our program and display the output. Coupled with SimAVR we are able to write sequences of input and trace how those inputs generate the output to verify correct functionality of our program. We have already used one GDB script (`commands.gdb`) to provide additional commands, now we will be writing some test cases in another script. There is a template in the `test` folder (`test/tests.gdb`)

Now to write some tests. The output on port B should be 0x01 when the input on PA0 is 1 and 0x02 when the input on PA0 is 0. All other pins on port A should not affect the value of port B. So let's write the following three tests.

1. PINA set to 0x00, output on PORTB should be 0x02
2. PINA set to 0x02, output on PORTB should remain 0x02
3. PINA set to 0x01, output on PORTB should be 0x01

```
test “PINA: 0x00 => PORTB: 0x02”
setPINA 0x00
continue 5
expectPORTB 0x02
checkResult

test “PINA: 0x02 => PORTB: 0x02”
setPINA 0x02
continue 5
expectPORTB 0x02
checkResult

test “PINA: 0x01 => PORTB: 0x01”
setPINA 0x01
continue 5
expectPORTB 0x01
checkResult
```

Now we will use the Makefile to compile and test your program.

```
$ make test
======================================================
Running all tests..."

Test 1:"PINA: 0x00 => PORTB: 0x02"...
passed.
Test 2:"PINA: 0x02 => PORTB: 0x02"...
passed.
Test 3:"PINA: 0x01 => PORTB: 0x01"...
passed.
Passed 3/3 tests.
======================================================
```

Exercises

After each exercise, copy your `source/main.c` into the turnin folder

```
$ cp source/main.c turnin/[cslogin]_lab2_part[#].c)
```

Each part should be demoed to a TA. If practical, demonstrate a working part to your TA before moving on to the next exercise (if students are waiting, you can move on, but demo as soon as possible). Normal exercises are worth 95% of your grade, "**Challenge**" exercises are to be completed if time permits (worth 5% each).

1. Garage open at night-- A garage door sensor connects to PA0 (1 means door open), and a light sensor connects to PA1 (1 means light is sensed). Write a program that illuminates an LED connected to PB0 (1 means illuminate) if the garage door is open at night.

PA0 = garage door sensor (input), PA1 = light sensor (input), PB0 = LED (output)

Input		Output
PA1	**PA0**	**PB0**
0	0	0
0	1	1
1	0	0
1	1	0

```
$ cp source/main.c turnin/[cslogin]_lab2_part1.c
$ cp test/tests.gdb turnin/[cslogin]_lab2_part1_tests.gdb
$ git commit -m "Completed part 1"
```

2. Port A's pins 3 to 0, each connect to a parking space sensor, 1 meaning a car is parked in the space, of a four-space parking lot. Write a program that outputs in binary on port C the number of available spaces (Hint: declare a variable "`unsigned char cntavail`"; you can assign a number to a port as follows: `PORTC = cntavail;`).

```
$ cp source/main.c turnin/[cslogin]_lab2_part2.c
$ cp test/tests.gdb turnin/[cslogin]_lab2_part2_tests.gdb
$ git commit -m "Completed part 2"
```

3. Extend the previous program to still write the available spaces number, but only to PC3..PC0, **and** to set PC7 to 1 if the lot is full.

```
$ cp source/main.c turnin/[cslogin]_lab2_part2.c
$ cp test/tests.gdb turnin/[cslogin]_lab2_part2_tests.gdb
$ git commit -m "Completed part 2"
```

4. (**Challenge**) An amusement park kid ride cart has three seats, with 8-bit weight sensors connected to ports A, B, and C (measuring from 0-255 kilograms). Set PD0 to 1 if the cart's total passenger weight exceeds the maximum of 140 kg. Also, the cart must be balanced: Set port PD1 to 1 if the difference between A and C exceeds 80 kg. Can you also devise a way to inform the ride operator of the approximate weight using the remaining bits on D? (Interesting note: Disneyland recently redid their "It's a Small World" ride because the average passenger weight has increased over the years, causing more boats to get stuck on the bottom).

   (Hint: Use two intermediate variables to keep track of weight, one of the actual value and another being the shifted weight. Binary shift right by one is the same as dividing by two and binary shift left by one is the same as multiplying by two.)

```
$ cp source/main.c turnin/[cslogin]_lab2_part2.c
$ cp test/tests.gdb turnin/[cslogin]_lab2_part2_tests.gdb
$ git commit -m "Completed part 2"
$ git push
```

* If you are worried about losing your work, you can push after each commit.

Submission

Each student must submit their source files (`.c`) and test files (`.gdb`) according to instructions in the lab submission guidelines.

```
$ tar -czvf [cslogin]_lab2.tgz turnin/
```

Don’t forget to commit and push to Github before you logout!