ENEL260 CA7: Functions and I/O

Ciaran Moore

"Each level is constructed by combining parts that are regarded as primitive at that level, and the parts constructed at each level are used as primitives at the next level. The language used at each level of a stratified design has primitives, means of combination, and means of abstraction appropriate to that level of detail."

– Abelson & Sussman, *The Structure and Interpretation of Computer Programs*

Simple assembly instructions to add, subtract, load, store, and branch, only get us so far. How can we build more complex systems that include function calls? And how can we get data in and out of the microprocessor?

In the last lecture, we studied a tiny C program and how that is executed on a microprocessor. In particular, we looked at the *execution cycle*, comprising at least a 5-step execution cycle:

1. **Fetch** the instruction
   - Fetch the instruction from the location specified by the program counter PC
   - Increment the program counter, PC $\leftarrow$ PC + 1
2. **Decode** the instruction
3. **Execute:** Get the operand(s)
4. **Execute:** Perform the specified operation on the operands
5. **Execute:** Store the result of the operation

The instructions that a RISC processor executes can usually be grouped into one of three categories:

1. *Arithmetic or logical*: such as add, subtract, AND, OR, compare, etc.
2. *Branching instructions*: choose between possible paths through the instructions[1] by changing the PC.
3. *Data transfer*: store data from a register to a location in memory, or load data into a register from a location in memory.

[1] "**if** <condition> do some stuff **else** do some other stuff"

You may have noticed that the instructions mentioned above don't include any kind of function call[2], or any input and output. In this lecture we'll take a look at how *function calls* fit into the machine model we've already seen. We'll also look at how machine instructions can be used to *interface with external hardware*.

[2] Even though we make much use of functions in languages like C.

Function Calls

Let's consider the tiny C program:

```
#include <stdint.h>

void exchange(uint8_t *a, uint8_t *b) {
   uint8_t tmp;

   tmp = *a;
   *a = *b;
   *b = tmp;
}

int main(void) {
   uint8_t ball = 33;
   uint8_t bat = 15;

   exchange(&ball, &bat);
   return 0;
}
```

As you can see, this program doesn't really do anything useful. But it does include a useful function, `exchange()`. Using pointers for the two arguments allows the function to exchange the contents of two bytes in memory. This might be used, for example, as part of a *sorting*[3] algorithm.

[3] There are a lot of different types of sorting algorithms. Many of them involve performing swaps of the bytes being sorted.

Function Calls and the Stack

The stack is a section of data memory (you may have heard it mentioned in the **C Programming** section). At any time the *Stack Pointer* (SP) register (16 bits on the AVR) contains the address of the next byte to be written and the stack grows downwards (towards lower addresses). Note that this is an arbitrary choice by the microcontroller designers to make the `PUSH` instruction post-decrement SP. Correspondingly, the `POP` instruction pre-increments SP (to access the last pushed item on the stack).

As we saw in the last lecture, the compiler actually uses rather a lot of machine instructions to execute such a tiny program (74, in this case), but we can identify some key instructions and look at them in detail to illustrate what happens when an instruction is executed. We'll again use a simplified version of the architecture, which emphasises the datapath, the memories and the major components associated with instruction execution. Make sure you look at the extra information in the last lecture about some of the details of the instructions.

Function Calls

The line `exchange(&ball, &bat);` informs the compiler that the function `exchange` is to be executed once with the given arguments. Once that has completed, execution should continue at the C statement which follows the call. Since the call must change the contents of PC, the address of the following instruction must be retained;

this is achieved by using the stack. Also required is a means of passing the values of the arguments (in this case 2 pointers) to the function and, in some cases, returning a value from the function; again the stack is involved. When variables are declared inside a C function (these are known as *local variables*), they are also stored on the stack (the single byte `tmp`, in this case). And, finally, it is necessary to preserve the values of some of the general purpose registers, since it's possible that they may be reused within the function execution. The combination of all this data is called a *stack frame*.

To summarise, the *stack frame* associated with a function call contains:

- contents of PC (address of the next instruction after returning)
- values of arguments being passed to the function
- value(s) of any return from the function
- variables local to the function
- the contents of registers which may be written to during the function execution

The Stack Frame

The stack frame assembled by the `avr-gcc` compiler for the call to the function `exchange` comprises 9 bytes as shown below:

address	contents	also called	notes
0x4FA	. . .		End of previous frame
0x4F9	PC(7:0)		Start of frame
0x4F8	PC(15:8)		
0x4F7	R28	Y (low byte)	Y is used to point to the stack frame
0x4F6	R29	Y (high byte)	(0x4F0) while the function is executing
0x4F5	`&a`	address low byte	1st argument
0x4F4	`&a`	address high byte	
0x4F3	`&b`	address low byte	2nd argument
0x4F2	`&b`	address high byte	
0x4F1	`tmp`		End of frame
0x4F0	. . .	'top' of stack	Next position on the stack

This stack frame is actually created by a series of instructions.

1. Prior to executing the `call` instruction that changes the PC to execute code associated with the function `exchange`, several instructions place the addresses of the `a` and `b` variables into R23:R22 and R25:R24.

```
e8: ce 01         movw  r24, r28
ea: 02 96         adiw  r24, 0x02 ; 2
ec: bc 01         movw  r22, r24
ee: ce 01         movw  r24, r28
f0: 01 96         adiw  r24, 0x01 ; 1
```

2. The `call` instruction (a 32-bit instruction) places the address of the next instruction after the `call` onto the stack, and sets the PC to the address of the `exchange` function.

```
f2: 0e 94 46 00   call  0x8c  ; 0x8c <exchange>
```

3. The first part of the `exchange` function (at address `0x08c`) sets up the rest of the stack frame by pushing the Y register onto the stack (saving the address of the previous stack frame), using `rcall` to allocate stack space for the 16-bit arguments[4], and using a `push` to allocate space for the 8-bit `tmp` variable.

[4] This may seem a little odd, but is a low-overhead way to set aside 16 bits of stack space at a time.

```
8c: cf 93         push  r28
8e: df 93         push  r29
90: 00 d0         rcall .+0       ; 0x92 <exchange+0x6>
92: 00 d0         rcall .+0       ; 0x94 <exchange+0x8>
94: 1f 92         push  r1
```

4. The stack setup is completed in the next few instructions, by copying the SP register into Y (making the address of the current stack frame available for indirect addressing) and then copying the values of function arguments from the R25:R24 and R23:R22 registers into the appropriate locations within the stack.

```
96: cd b7         in  r28, 0x3d ; SP low byte
98: de b7         in  r29, 0x3e ; SP high byte
9a: 9b 83         std Y+3, r25  ; 0x03
9c: 8a 83         std Y+2, r24  ; 0x02
9e: 7d 83         std Y+5, r23  ; 0x05
a0: 6c 83         std Y+4, r22  ; 0x04
```

At the end of `exchange`, the local variables and arguments are popped from the stack, the Y register is restored to the value it had prior to the function call (moving to the previous stack frame), and `ret` is called to reset the PC back to the next instruction in `main`:

```
c6: 0f 90         pop r0
c8: 0f 90         pop r0
ca: 0f 90         pop r0
cc: 0f 90         pop r0
ce: 0f 90         pop r0
d0: df 91         pop r29
d2: cf 91         pop r28
d4: 08 95         ret
```

The figures on the next few pages (Fig. 1, Fig. 2, Fig. 3) show the behaviour of the `call`, `push`, and `ret` instructions.

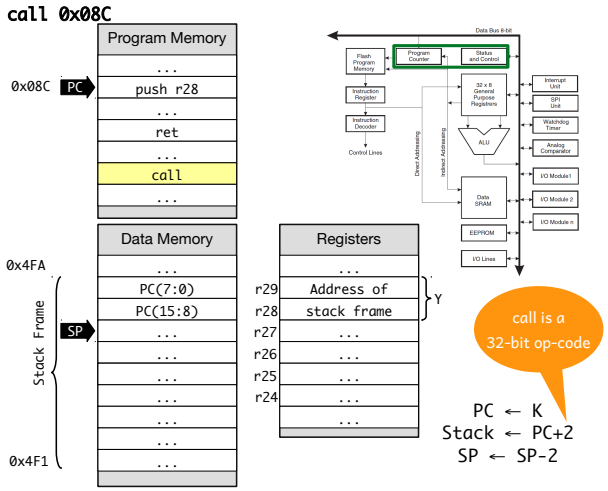

Figure 1: CPU execution cycle for instruction `call 0x08c`, *Long Call To A Subroutine*.

call

Syntax	Operands	Operation
`call k`	$0 \leq k \leq 64K$	STACK ← PC + 2
Control registers	Opcode	Flags
PC ← k SP ← SP - 2	1001 010k kkkk 111k kkkk kkkk kkkk kkkk	none

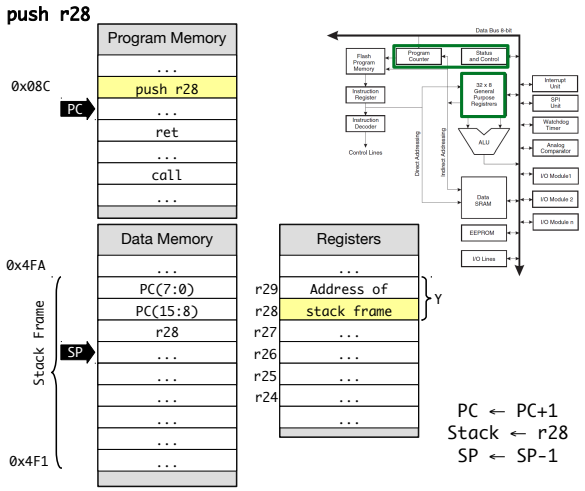

Figure 2: CPU execution cycle for instruction `push r28`, *Push Register on Stack*.

push

Syntax	Operands	Operation
`push Rr`	$0 \leq r \leq 31$	STACK ← Rr
Control registers	Opcode	Flags
PC ← PC + 1 SP ← SP - 1	1001 001r rrrr 1111	none

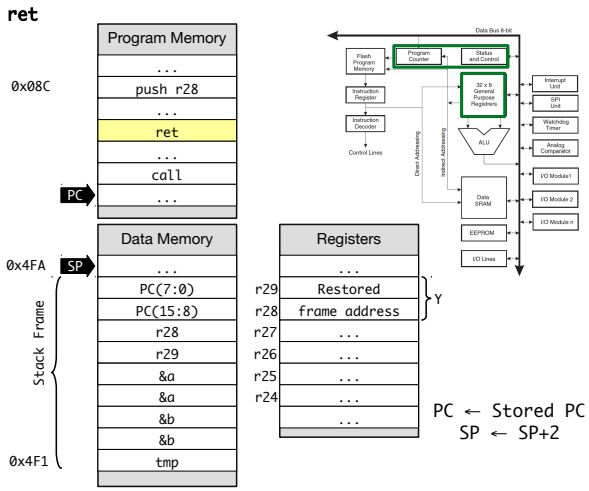

Figure 3: CPU execution cycle for instruction `ret`, *Return from Subroutine* (at the completion of `exchange`).

`ret`

Syntax	Operands	Operation
`ret`	none	PC ← STACK
Stack pointer	Opcode	Flags
SP ← SP + 2	1001 0101 0000 1000	none

General Purpose Input Output (GPIO)

As you will have learned in the Embedded section of the course, the AVR microcontroller has a set of General Purpose Input Output (GPIO) ports, each with 8 pins.

Each of those ports is primarily controlled by three registers:

- **DDRx** Sets the Data Direction for Port x, i.e., is the pin configured for input or output.
- **PORTx** Sets the output state of each pin for Port x, i.e., HIGH = 5 Volts or LOW = 0 Volts.
- **PINx** Gets the input state of each pin of Port x, i.e., is the input HIGH = 5 Volts or LOW = 0 Volts.

Fig. 4 shows the logic circuit that is connected to each of the pins. Actually, although it looks complicated enough, what appears is somewhat simplified from the complete circuit. Note that the three registers `DDRx, PORTx and PINx` are shown and they are made of edge-triggered D-type flip flops.

In Fig. 4, there are several symbols that we have not studied before. Three of these are detailed in Fig. 6. The most important of the three shown is (b) the *tristate buffer*; when the control input is LOW, the output is OFF (high impedance - independent of the buffer input). The remaining "strange" symbol in the lower left of Fig. 4 is a transmission gate (the symbol is known as a 'bowtie').

Figure 12-2. General Digital I/O(1)

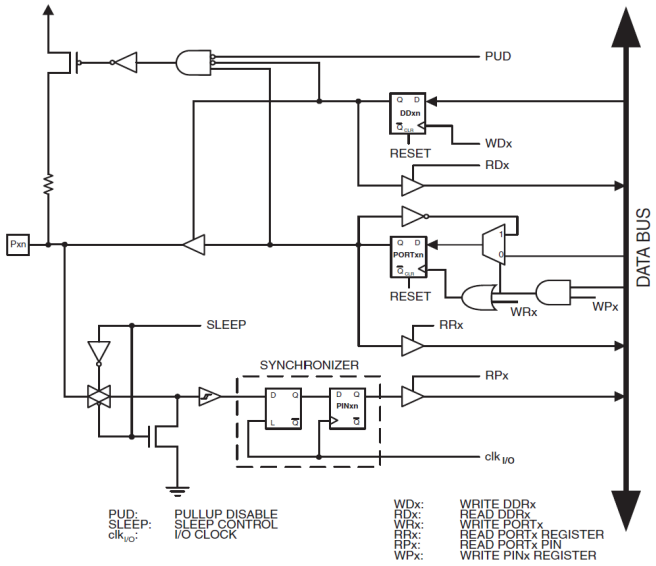

Figure 4: The logical circuit for a single GPIO pin on the ATmega32U2 microcontroller.

The pair of flip-flops labeled *"synchronizer"* is designed to avoid problems with input signals changing right on a clock edge. The first flip-flop is "level triggered": it will accept an input when the clock signal is high, and will latch the most recent input value when the clock signal is low. The second flip-flop is "edge-triggered", and picks up the latched output of the first flip-flop when the clock transitions from low to high (has a rising edge).

Table 12-1. Port Pin Configurations

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Figure 5: Primary control truth table for the GPIO pin shown in Fig. 4.

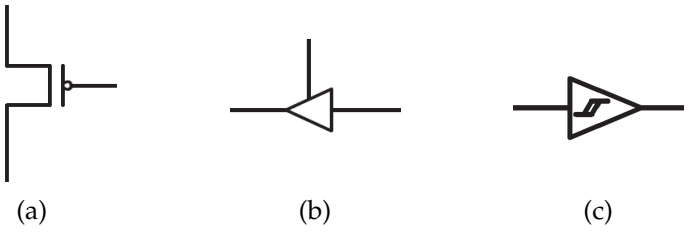

Figure 6: Individual components which appear on the circuit diagram for the GPIO pin: (a) electronic switch (gate = LOW, switch is ON; (b) tristate buffer (control = LOW, output is OFF); and (c) Schmitt buffer (threshold $\uparrow$ > threshold $\downarrow$).

You are not expected to know anything about interrupts and ISRs in ENCE260.

Looking Deeper: Interrupt Service Routines An *interrupt service routine* (ISR, also called an interrupt 'handler') is a special type of function. An *interrupt* is a signal used to break the cycle of execution to allow an important urgent task to be carried out. Examples are when an external signal is received that needs to be dealt with quickly, when the result from an analog-to-digital conversion becomes ready, or when an internal timer reaches a particular significant value. An ISR is a special function that is called in response to the occurrence of an interrupt. An ISR differs from a normal function call in the following ways:

- the compiler doesn't know when the function is to be called (interrupts can occur at arbitrary times, and the ISR is not part of the normal sequence of program execution)
- the contents of the status register (SR) need to be stored in the stack frame
- the function takes no arguments
- the return type of the function must be `void` (there's no where to return a value to, since the ISR is not called from the main program)