Linnéuniversitetet
Kalmar Växjö

Linnaeus University

Faculty of Technology – Department of Computer Science

1DT301 - Computer Technology Lab 2

Student: Hussam Aldeen ALKHAFAJI
Student ID: ha223cz@student.lnu.se

1. **Explain the process of triggering an interrupt in AVR, from start to finish.**[1]

Interrupts in AVR must be initialized for them to become active and useable.

This initialization is a two-step process that is similar in almost all AVR processors, keeping in mind that the interrupt sources and vectors (including registries names) will depend on the module used.

The first step consist of unmasking the interrupt to be used.
This is done by setting the bit that corresponds to the interrupt on the specified registry to HIGH.
This registry depends on the module used. By example, for the Atmega16, the registry's name is GICR, while in the Atmega2560 the registry is called EIMSK.

The second step consist of enabling interrupts globally for the active project. This is basically achieved by injecting the assembly instruction (sei), either by calling the function or by specifying to the compiler that you want to inject this assembly instruction by #asm("sei").

Those two steps are essential as the processer will logically AND all those bits for the interrupt to work.

There's an additional optional step that consist of setting the behavior of the interrupt by modifying the EICRA or the EICRB registries (for atmega2560). We can have it so when the interrupt is falling edge (or rising edge) it activates if we wish so, or we can leave it as default.

Given that the interrupt service routine is already written, the interrupt is now ready to be used and triggered by the specified pin.
When it gets triggered, it'll execute the corresponding ISR function.
(Embedded C Programming and the Atmel AVR, 2nd Edition n.d.)

2. **Explain what an opcode is, identify the opcode for the assembler instructions ADD and SUB.**

An opcode is the portion of machine language instruction (binary) that specifies the operation to be executed. It's an abbreviation of operation code (aka instruction machine code). Most instructions specify the data to be processed by the operation (operands) alongside the opcode itself.

The opcode for ADD:

0000 11rd dddd rrrr

With d being the destination register address and r being the source register address.
Rd = Rd + Rr.

The opcode for SUB:

0001 10rd dddd rrrr

With d being the destination register address and r being the source register address.
Rd = Rd - Rr.
(Opcode n.d.)

3. What is the difference between assembly and machine code?

The main difference is machine code is executed directly by the computer since it consist of strings of binary numbers that represents specific information.
While the assembly code requires and assembler to convert it or assemble it to machine code to be comprehensible for the computer.

In more detail,
Machine code consist of zeros and ones which are directly comprehensible by the computer since it's an electronic machine built with thousands and thousands of transistors that are they themselves built by logic gates.
It's not comprehensible by human beings at all.

Assembly, is an instruction language that is intermediate between high level programming languages (C++, python) and low level machine code.
It's comprehensible by human beings but is much more difficult to master given how close to the hardware it is.

Using it requires a good understanding of the architecture of the used machine because the programmer have to handle everything from memory to registers to CPU by himself.
It also needs an assembler to translate it to machine code.
(opcodeVsmachincode n.d.)

4. **Explain the stack discipline -- why is it useful**

The stack refers to two things, either the data structure, or the SRAM in the AVR microcontrollers which behave like a stack data structure.
There are two stacks in the SRAM of the AVR, the system stack which starts from the top of the memory and utilizes the memory by going downwards.
The processor uses this stack to keep information that doesn't belong in registers, like function return addresses,
intermediate results from calculations, and any other short-term temporary data storage.
This is very useful as it permits the usage of, by example, recursive functions.
The other stack is the data stack and it starts below the system stack and works its way down through memory, in a similar way to the system stack, and is used to store
temporary data, such as the local variables used in a function.
(Embedded C Programming and the Atmel AVR, 2nd Edition n.d.)

5. **VG Task: decompiler solution writeup.**

The solution is simple, so it'll be described in steps;

1- Receive machine code as input
2- Take each instruction, separate the 4 MSBs using bit masking, and pass the result alongside the full instruction to the function chooseOpcode().
3- The function chooseOpcode() uses a switch statement to decide which operation is to be executed, after a match is found, it calls the corresponding function that translates the machine code to assembly.
4- Each function handles the machine code depending on the instruction it servers. Ultimately, all of them use bit masking

techniques to extract the operand(s) and display them according directly to stdout.

Bibliography

1- *Embedded C Programming and the Atmel AVR, 2nd Edition.* n.d.

2-Opcode. n.d. https://en.wikipedia.org/wiki/Opcode.

3-opcodeVsmachincode. n.d. https://www.differencebetween.com/difference-between-machine-language-and-vs-assembly-language/.