ENEL260 CA6: Executing Programs

Ciaran Moore

With knowledge of registers, ALUs, and Finite State Machines, we now have enough information to understand how a CPU works.

"A computer will do what you tell it to do, but that may be much different from what you had in mind" – Joseph Weizenbaum

A MICROPROCESSOR (OR CPU) CONSISTS OF at least the following components:

1. A set of general purpose registers (sometimes called a GPR *file*)
2. A set of control registers[1]: the program counter (PC), the status register (SR) and the stack pointer (SP)
3. An arithmetic logic unit (ALU)
4. A control unit: a large FSM that controls what happens and when. The state of the FSM is determined by the program instructions and the control registers. The outputs of the FSM control:
   - What data (from registers) is presented to the ALU, what ALU function is performed, and where the ALU results go
   - Signaling on the address and control buses
   - Changes to values stored in the control registers (i.e., changes in FSM state)

[1] The instruction register can be included too

Fig. 1 is a view of the architecture of the ATmega32U2, the microcontroller at the heart of the fun kit (UCFK). The coloured annotations identify important sections in the design. You'll find the components listed above, as well as some other components that we have discussed a little in earlier lectures.

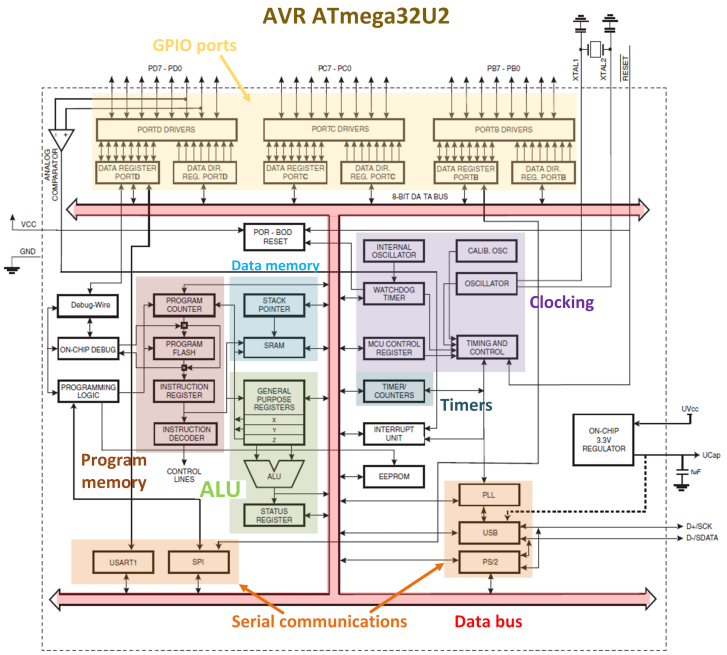

Figure 1: CPU Architecture

The Execution Cycle

AT A HIGH LEVEL, all CPUs go through three major phases of execution:

1. **Fetch** the instruction from memory
2. **Decode** the instruction
3. **Execute** the instruction

These phases are repeated indefinitely.

The execution process is managed by the Control Unit. The main input to the Control Unit is the fetched instruction, and its states are the various steps of the execution cycle. Thus the Control Unit determines when and how the instruction fetch, decode, and execute processes take place.

Most machines use a program counter to keep track of which instruction should be executed, and provide instructions that can operate on stored data. In such machines, the three phases described above are implemented as a 5-step[2] execution cycle:

[2] Actually "at least 5 steps". There may be additional steps, depending on the CPU design.

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 Instruction Set

THE INSTRUCTIONS THAT THE MACHINE EXECUTES are a set of binary codes, known as an *Instruction Set*. Like the coded list of vegetable types that we looked at when first considering codes, the coding of the machine instructions must be agreed upon in advance[3].

[3] The programmer and the machine must both agree on what each specific pattern of bits "means".

Instruction Sets are often unique to a type of microprocessor, or to a family of microprocessors. Because different Instruction Sets have different codes, or assign different meanings to codes, machine code for one machine cannot be expected to run on machines of different types: Intel I3 machine code won't run on a ARM7 processor.

Typical machines have many tens to several hundreds of instructions. The instructions 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[4]
3. *Data transfer*: store data from a register to a location in memory, or load data into a register from a location in memory.

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

Broadly speaking, instruction sets can be divided into two classes:

Most modern machines are of the RISC variety, as this can lead to faster and more efficient CPU designs.

- *Reduced Instruction Set Computers (RISC)* only allow arithmetic and logical instructions to access data from the CPU registers. These machines have data transfer and memory addressing instructions which are separate from the arithmetic and logical instructions.
- *Complex Instruction Set Computers (CISC)* allow the arithmetic instructions to access data from CPU registers, or directly from memory. These machines often have few or no separate transfer and addressing instructions, rather the addressing is a sub-instruction to the arithmetic instruction.

The design philosophy of RISC is that each instruction is simple and does a single task, such that almost all tasks can be completed in the same time frame (1 instruction per cycle).

CISC instructions allow complicated operations to be specified using a single machine instruction. But the complexity of the operations means that the execution time of instructions is highly variable and can last for many clock cycles.

Instruction Execution: Some Examples

The AVR ATmega32U2 is a RISC processor with a Harvard architecture. On the next few pages, let's step through of the execution of a few instructions on the ATmega32U2.

Consider the tiny C program:

Because the variables `index` , `sum` , and `array` are all declared locally to the `main()` function they will all be located in a part of the data memory called *the stack*.

```
int main(void) {
    int index, sum, array[4] = {-1.0, 3, 5, -2};

    sum = 0;
    for (index = 0; index < 4; index++) {
        sum = sum + array[index];
    }
    return 0;
}
```

For our example execution, we'll use a *simplified model* of the ATmega32U2 architecture (Fig. 2) that emphasizes the datapath, the memories, and the major components associated with instruction execution.

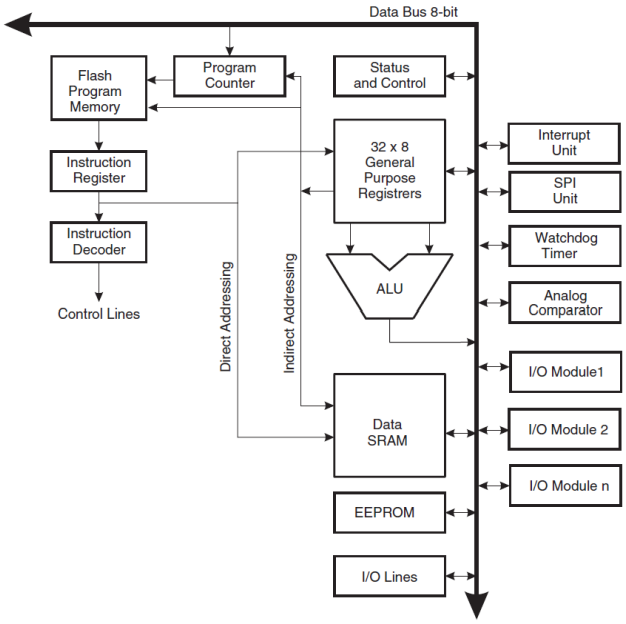

Figure 2: Simplified architecture of the ATmega32U2

A few additional pieces of information[5] will help in interpreting what's going on:

- All AVR instructions occupy 16 bits
- Three pairs of registers are used for addressing data items: X = r27:r26, Y = r29:r28, Z = r31:r30
- In AVR assembly language, each instruction is written as: **mnemonic destination, source**
  - *Mnemonics* are abbreviated names for instructions, e.g. `ldd`, `adiw`, etc.
  - The *source* may be a location or an instruction operand

[5] This sort of information is typically available in microcontroller datasheets, programmer's manuals, instruction set manuals, or other related documentation. For AVR assembly, see https://www.microchip.com/webdoc/avrassembler/.

The AVR compiler actually uses a lot[6] of machine instructions to execute this tiny program. We'll look at just a few key instructions in detail, to illustrate what happens when an instruction is executed. The short instruction sequence we'll look at is executed at the end of the an iteration through the `for` loop:

[6] 87, in fact!

```
// index++
ldd  r24, Y+1  // Load the value of 'index' from the stack
ldd  r25, Y+2  // into r24 and r25
adiw r24, 0x01 // Add 0x01 to r25:r24
// Store r25:24 to stack (not shown here)

// Test for index >= 4
sbiw r24, 0x04 // Subtract 0x04 from r25:r24
brlt .-56      // Go back 56 bytes (28 instructions) if result
    was negative
```

`Y` is a pair of 8-bit registers that together contain the address of the top of the current "stack frame". Since the stack grows *downwards* through memory, `Y+1` is the location of the first byte added to the stack, `Y+2` the second, and so on.

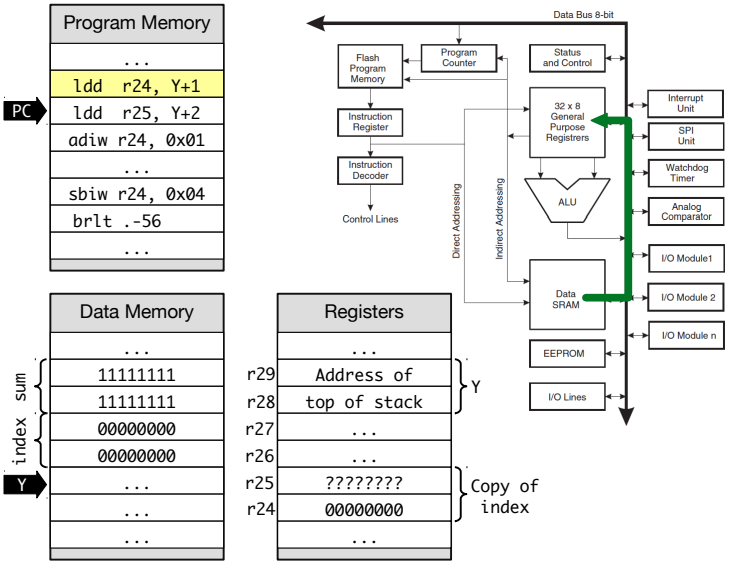

Figure 3: CPU execution for instruction `ldd r24, Y+1`, *Load Indirect from Data Space to Register using Index Y*.

`ldd`

Syntax	Operands	Operation
`ldd Rd, Y+q`	$0 \leq d \leq 31, 0 \leq q \leq 63$	Rd $\leftarrow$ (Y+q)
Program counter	Opcode	Flags
PC $\leftarrow$ PC + 1	10q0 qqod dddd 1qqq	None

In Fig. 3:

- The instruction has been fetched and decoded, so PC is already pointing to the next instruction.
- The contents of register pair Y (r29:r28), which contains the address of the top of the stack frame, is used to find the location of `index`
- The first `ldd` instruction increments Y by 1, to access the LSB of `index` and load it into r24. The next `ldd` will load the MSB of `index` into r25.

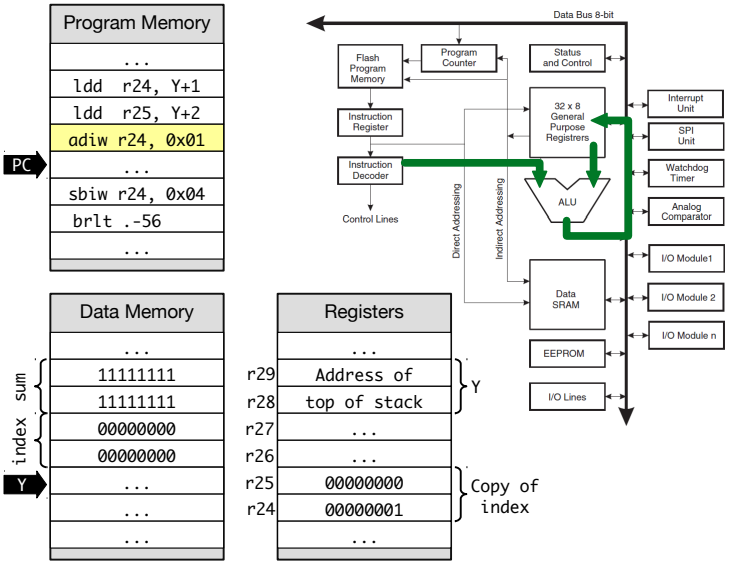

Figure 4: CPU execution for instruction `adiw r24, 0x01`, *Add Immediate to Word*.

adiw

Syntax	Operands	Operation
`adiw Rd+1:Rd,k`	$d \in \{24, 26, 28, 30\}, 0 \leq k \leq 63$	Rd+1:Rd ← Rd+1:Rd+k
Program counter	Opcode	Flags
PC ← PC + 1	1001 0110 kkdd kkkk	S, V, N, Z, C

In Fig. 4:

- The instruction has been fetched and decoded, so PC is already pointing to the next instruction.
- The instruction operates on an entire 16-bit word (hence the name *Add Immediate to Word*).
- The operands sent to the ALU are the contents of r25:r24, and a constant value (0x01) that is encoded in the instruction op-code.
- The result (1) is placed back into r25:r24.

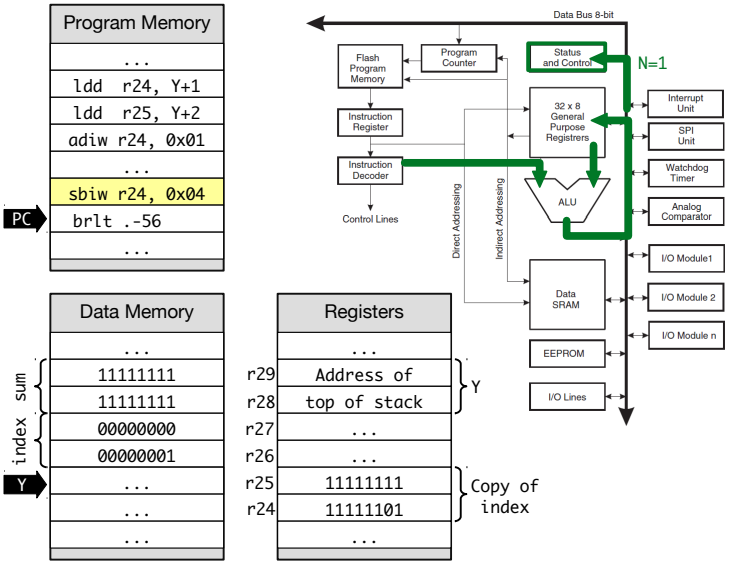

Figure 5: CPU execution for instruction `sbiw r24, 0x04`, *Subtract Immediate from Word*.

`sbiw`

Syntax	Operands	Operation
`sbiw Rd+1:Rd,k`	$d \in \{24, 26, 28, 30\}, 0 \leq k \leq 63$	Rd+1:Rd ← Rd+1:Rd-k
Program counter	Opcode	Flags
PC ← PC + 1	1001 0111 kkdd kkkk	S, V, N, Z, C

In Fig. 5:

- The instruction has been fetched and decoded, so PC is already pointing to the next instruction.
- We've skipped over some instructions that stored the value of r25:r24 back to `index` (note the change in data memory).
- The instruction operates on an entire 16-bit word (hence the name *Subtract Immediate from Word*).
- The operands sent to the ALU are the contents of r25:r24, and a constant value (0x04) that is encoded in the instruction op-code.
- The result (-3) is placed back into r25:r24.
- Since the result is negative, the `N` flag is set in the status and control registers.

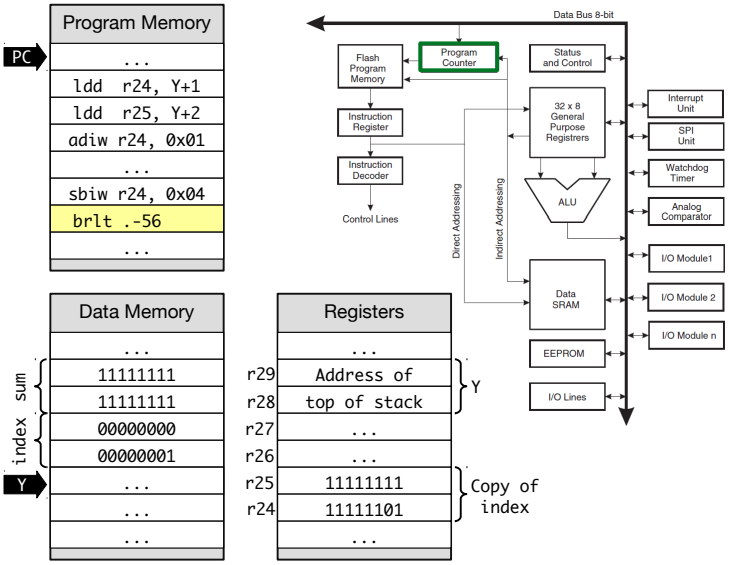

Figure 6: CPU execution for instruction `brlt .-56`, *Branch if Less Than (Signed)*.

`brlt`

Syntax	Operands	Operation
`brlt k`	$-64 \leq k \leq +63$	Is result of CP, CPI, SUB or SUBI < 0?
Program counter	Opcode	Flags
PC ← PC + 1 (false) PC ← PC + k + 1 (true)	1111 00kk kkkk k100	None

In Fig. 6:

- The `brlt` instruction checks the `S` flag.
- Since the `S` flag is set, -28 is added to the program counter (PC). This returns execution to the start of the loop.
  - Why -28 instead of -56? The instruction that we're seeing in this example doesn't actually directly contain the `k` operand. Instead, the compiler has generated some code that uses a feature of the GNU assembler to compute a value based on the current byte address (represented by '`.`') and a number of *bytes* (rather than 16-bit instructions), and uses that to generate the appropriate value of `k`. If you were writing the same instruction by hand you might instead write `brlt -28` (note the lack of a '`.`').

Programming Languages

EARLY PROGRAMMING required literally entering the instructions to the machine as strings of binary or hexadecimal codes. This was very time consuming and error prone.

A natural progression from binary instructions was to encode instructions as "assembly"[7] codes. Assembly languages allowed the programmer to use a mnemonic text string or word (e.g. "ADC" and "STS") to represent an instruction. A program written in assembly language could be mechanically translated into the corresponding binary or hexadecimal instructions[8]. Along with this process came the ability to group together instructions with the same functionality, differing only in their *operands* (making programs easier to understand), and to define "macro instructions" that corresponded to multiple machine instructions (making programs shorter).

Using assembler is somewhat more efficient than writing programs directly in machine codes, but is still relatively primitive: most assembly codes correspond directly to individual machine instructions[9], and the only control structure is some kind of *jump* instruction that is essentially the equivalent of a "goto" instruction. The next generation of languages[10], which include *C*, provided enormous efficiency gains because they provided complex data and control structures: `if-then-else` , `while-do` , `switch-case` , `struct` ures, etc. These made it possible to focus less on details of *how* the computer is doing its work (e.g., copying a value from the stack into a pair of registers), and more on *what* the program was supposed to be doing (e.g., summing an array of numbers).

[7] So-called because they were "assembled" into a binary program.

[8] Either by hand, or using an "assembler" program.

[9] So writing complex algorithms requires thinking about not just the algorithm, but the movement of individual pieces of data between different registers and other low-level details, as we saw in the example of instruction execution above.

[10] "3rd generation languages", or 3GLs