CSC 230 Notes

Liam Shatzel

January 2023

Contents

1 Computer Systems

- What is a computer?
  - Device that accepts input
  - Stores/retrieves data
  - produces output
- Main components of a computer system
  - Hardware (physical devices/ hardwired logic)
  - Software (instructions and data)
  - Device Drivers (connect the hardware and software). Device drivers are also a part of software.

1.1 Hardware:

- Hardware covers the physical components of the computer system
- it also encapsulates wired connections and predefined functionality
- In other words, hardware always stays the same, each part has a specific task and functionality and cannot change.

1.1.1 Components

- CPU - central processing unit
- Memory - RAM (volatile) random access, ROM (non-volatile) (Read only)
- Storage: HDD, SDD, DVD
- Input devices: Keyboard, mouse, touchpad, buttons
- Output devices: Screen, printer, network, motors
- Communication Devices: Network, infrared, Bluetooth...
- Main BUS cable
  - BUS is made up of multiple wires.
  - Connects all points of the computer and carries info from CPU to various parts of the computer.
  - CPU and memory must be high speed
  - Peripherals are connected through controllers in order to not slow CPU down.
  - different connections between the bus (Data, Address, Control)
  - Data: Carries the actual data
  - Address: Distinguishes which address we want to talk to
  - Control: Control some operation, whether it is read or write.

1.2 Software:

- Programs or lists of instructions
- These can be changed at any time and this is where the soft part of the name "software" comes from.

1.3 Peripherals:

- Anything that isnt part of the core components of the computer. i.e. mouse, keyboard etc...

2 Basic Processor Architecture

Sections of the CPU:

- Control Unit (CU)
- Arithmetic / Logic Unit (ALU) (add, sub, boolean logic)
- Registers

2.1 CPU Execution Cycle

- fetch: reading instructions from memory (external to CPU) and loading to registers inside CPU
- decode: take the instructions that we just read and determine its meaning (what kind of operation should happen) (happens in control unit CU) (each instruction has an opcode (operation code))
- execute: actions indicated through decoding are then performed. Done in the ALU (arithmetic logic unit)
- final step: execute/store (writes back to main memory AKA writeback)

Note: operation codes (opcodes): binary number that is unique and identifies an operation (everything is binary this low down, needs to be broken down into smaller chunks to be understood).

2.2 Timing

- Architectures differ in how they use timing.
- One approach is to fit each fetch, decode, execute cycle into one clock cycle.
- the faster the clock speed the faster the operation, the clock speed itself depends on the hardware.
- 1000 Hz = 1 kilohertz = 10^3 Hz = 1 KHz
- 1000000 cycles per second = 1 megahertz
- 1000000000 cycles per second = 1 gigahertz

3 CPU Architecture

- CPU Coordinates operations of every computer system (e.g. process data in small units, arithmetic and logical ops, store and retrieve data, communicates with peripherals.)
- each CPU family has its own machine language
- consists of:
- Control Unit: CU
- Arithmetic Logic Unit: ALU
- Registers: intermediate memory

3.1 Control Unit

What it does:

- Decides what CPU components are to be used next
- Responsible for guidance on fetch, decode and execute instructions (signals who does what)
- Acts like a director causing data to flow to right places in CPU
- tells ALU to add multiply AND, OR, etc
- capable of branching, choose among several flows of control
- Communicates between RAM and registers and loads data into registers
- system clock does the overall synchronization.

3.2 Arithmetic Logic Unit

- Built-in operations within the hardware for int, float and logic ops
- CU gets data to ALU and tells it which operations to perform
- sometimes instructions are called microinstructions
- Operations it performs: Add, Sub, Mult, divide ...
- Logical Operations: comparisons, equal zero, check negative

3.3 Registers:

- Memory locations inside the CPU
- Designed for use with instructions being performed
- Hold operands and results from ALU e.x. ADD(a,b): a, b are operands
- e.g. CPU adds two numbers, one operand is in one register, one in another register addition is done and results are stored in possibly another register.
- Registers are expensive i.e. AVR has 32 registers 1 byte each

4 Memory

- Memory is normally a sequence of bytes
- byte = 8 bits
- 2 bytes = word
- processor keeps track of address of next instruction (next address to be fetched)
- the program counter (PC) AKA instruction pointer (IP) keeps track of this.
- After the fetch PC is updated to address of next instruction (usually by a fixed increment)
- the PC is just a register which we read or write from. Writing to the PC will cause it to jump to the instruction specified by the address.
- memory has two main uses: containing data (variables constants), containing code

4.1 Von Neumann Architecture

- Same memory is used for code and data
- AKA Princeton architecture
- Von Neumann is what all modern computers use.

4.2 Harvard Architecture

- Code is in one memory system and data is in a separate system
- Good for embedded systems, arduino uses Harvard.
- potentially faster performance by using different buses for instructions and data
- two different kinds of registers for two different kinds of memory

5 Number Systems and Conversions

dec	bin	oct	hex
0	0b0	0o0	0x0
1	0b1	0o1	0x1
2	0b10	0o2	0x2
3	0b11	0o3	0x3
4	0b100	0o4	0x4
5	0b101	0o5	0x5
6	0b110	0o6	0x6
7	0b111	0o7	0x7
8	0b1000	0o10	0x8
9	0b1001	0o11	0x9
10	0b1010	0o12	0xA
11	0b1011	0o13	0xB
12	0b1100	0o14	0xC
13	0b1101	0o15	0xD
14	0b1110	0o16	0xE
15	0b1111	0o17	0xF
16	0b10000	0o20	0x10

5.1 Positional Notation:

$$7409.01 = 7 \times 1000 + 4 \times 100 + 9 \times 10 + 0 \times \frac{1}{10} + 1 \times \frac{1}{100}$$

Abstraction:

$$d_3 \times B^3 + d_2 \times B^2 + d_1 \times B^1 + d_{-1} \times B^{-1} + d_{-2} + B^{-2}$$

where B is the base and d is the digit at a given position.

In the example above $B = 10$ because we have 10 symbols that we are able to use. They start to be recycled after the first 10 digits are represented.

Generalized formula:

$$\sum_{i=-m}^{n-1} d_i B^i$$

In this case n is the number of digits to the **left** of the radix point (decimal point in example).
m is the number of digits to the **right** of the radix point.

5.2 Base Conversions

prefixes:

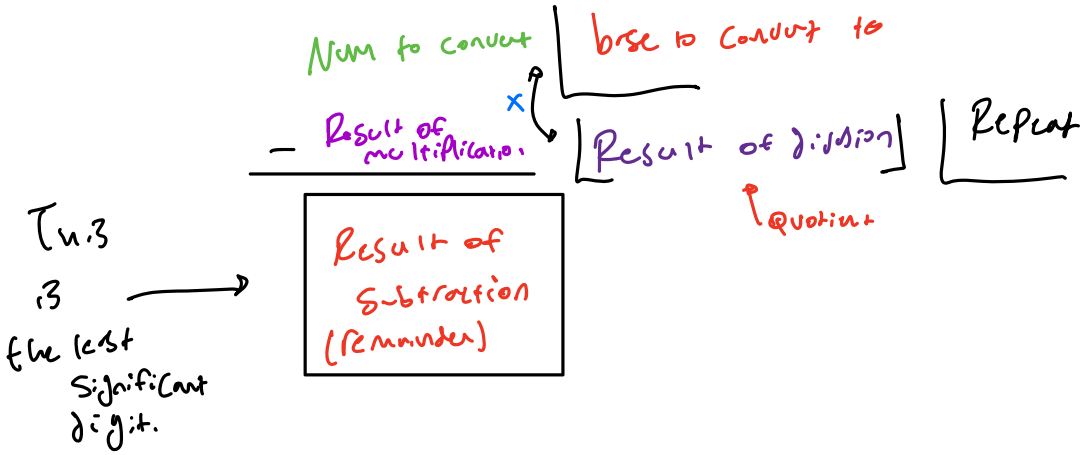
- binary: 0b
- hex: 0x
- octal: 0o

Techniques:

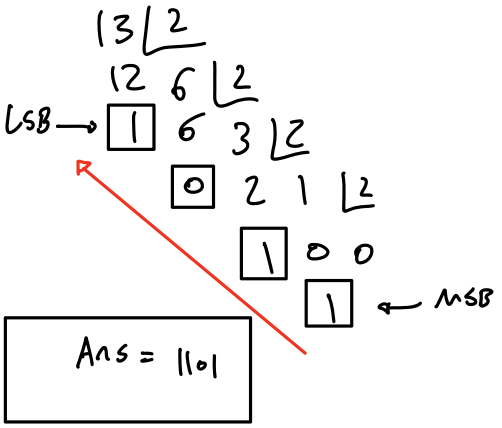
- Binary → hex: group in fours and convert (starting from right).
- Binary → octal: group in threes from right and convert.
- Hex → binary: convert each digit directly to 4 bits.
- Octal → binary: convert each digit directly to 3 bits.
- Hex ↔ octal: convert to binary then convert to wanted rep.
- Decimal → oct/bin/hex: repeated division algorithm (see 1).
- oct/bin/hex → decimal: polynomial evaluation algorithm (see above).

Horners Algorithm:

- An iterative algorithm to reduce the number of arithmetic operations needed to convert bases.
- Rather than having to raise our base B to a power multiple times, we can keep track of our current sum and multiply by the base.
- Similar to how we can append a zero to a decimal number and then add a digits to the ones place to create a larger number
- e.g. 512:
    1. 5 * 10 = 50
    2. 50 + 1 = 51
    3. 51 * 10 = 510
    4. 510 + 2 = 512 (as wanted)


Num to convert
base to convert to
Result of multiplication
Result of division
Repeat
Quotient
Result of Subtraction (Remainder)
The last Significant digit.

Ex: 13 → Binary


13 | 2
12 6 | 2
LSB → 1 6 3 | 2
0 2 1 | 2
1 0 0
1 ← MSB
Ans = 1101

5.3 Twos Complement

Twos complement lets us represent negative numbers in binary.

- The first idea to represent negative numbers that comes to mind is using the most significant bit as the sign bit, with 1 meaning negative and 0 meaning positive
- This is called ones complement and has a main problem that prevents its use.
- The main problem is that there are two representations of zero:
  - 0b0000
  - 0b1000
- To solve this we use twos complement.
- The complement of a binary number is the inversion of its bits.
- We invert the bits when we add to cause a rollover in the number we are trying to represent.
  - For example if we have a two-place number system (00-99) what would we add to 10 to get 00? To cause a rollover, we would add 90 such that 10+90 = 00, dropping the 1 (this is a form of subtraction).
- With this in mind for twos complement we invert the bits and add one (to account for the carry).

5.4 Binary-number Terminology

- bit = binary digit
- nibble = 4 bits
- byte = 8 bits
- word = 2 bytes (16 bits)
- double word = 4 bytes (32 bits)
- least significant bit (right-most bit)
- most significant bit (left-most bit)

5.5 AVR Arithmetic

5.5.1 Binary Addition

- Addition is performed on pairs of bits
- On the hardware level there are adders
- half-adder: takes two bits and gives the result of addition along with a carry (if there is one)
- full-adder: takes three bits, two bits to add along with a possible carry from a previous addition
- therefore an ADD instruction (between two 8 bit registers) requires 7 full-adders and at least one half adder (for the first two bits)

A	B	C	Sum	Carry
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

```
Boolean expression equivalents:

Sum = A XOR B XOR C
Carry = (A AND B) OR ((A XOR B) AND C)
```

A	B	Sum	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

```
Boolean expression equivalents:

Sum = A XOR B
Carry = A AND B
```

6 AVR Assembly

6.1 Arduino Specs

Arduino Mega 2560 (uses the ATmega 2560 chip)

- Harvard Architecture (data memory is separate from the code memory) **also good because we can load programs into flash memory without having to reupload later*
- RISC (Reduced Instruction Set Computer)
- Instructions are usually 16-bits in length (word)
- General purpose registers R0 - R31 (8 bits = 1 byte each). *register addresses are 5 bits in length ($2^5 = 32$)*
- PC register is 17 bits wide, this is because the processor has 256 kilobytes of flash memory which is 128 kilowords $\rightarrow$ 128,000 words, which are each a distinct address. Therefore we need $2^{17} = 131,072$, (as the $2^{16} = 65,536$ is the next smallest val and not enough).

6.2 Machine Language

- Machine instructions consist of: opcodes and operands (usually in the form of register addresses, or the data itself)
- Usually occupy some consecutive number of bytes.
- Program Counter (PC) is used here and contains the address of the next instruction code to be fetched.

6.3 Encoding Opcodes

```
LDI r16, 0b00000001
;the opcode for LDI is
;1110 | KKKK | dddd | KKKK
```

In this case the K's refer to the constant being loaded into the register.
The first 4 K's refer to the upper nibble of the byte being loaded
The last 4 refer to the lower nibble
the d's refer to the register numbers, since we only use the registers 16-31 for this operation we can cut off the MSB so that the byte fits in the opcode.
So:

1110	KKKK	dddd	KKKK

would turn into

1110	0000	0000	0001

Notice how the leading bit is cut of in the dddd column. This is equivalent to subtracting 16 from our destination register and representing that in binary. 16-16 = 0 = 0b0000

6.3.1 Endianness of Registers

The opcodes in the code memory may look a little different from what is expected. This is because the ATmega chip uses little-endian representation.

Little Endian
Little endian is when the "little end" of a word is at swapped with the "big end". More precisely the byte on the LSB side gets put in front.
Take the opcode above for example:
High byte : Low byte
11100000 : 00000001
In hex: 0x01E0
Putting this in little endian form gets us
11100000 : 00000001
finally translating into hex: 0xE001

Big Endian

Big endian would just have the representation stay the same. The "big end" stays at the front.
11100000 : 00000001
In hex: 0x01E0

6.4 Program Termination

```
loop:
    RJMP loop
```

- We create an infinite loop at the end of our program to prevent processing garbage instructions.
- Using RJMP instruction
  - relative jump as opposed to absolute jump (JMP)
  - relative jump jumps relative to where it currently is, all it has to do is add to the PC
  - absolute jump, jumps to a global position
- Action of RJMP PC = PC + 1 + k
- where the argument in the opcode is the value k
- in order to create an infinite loop we have to step the program counter back, therefore our k has to be = -1
- so the opcode must be encoded using 2's complement.
- RJMP has a 16-bit opcode, with 12 of those bits left to the constant and -1 in 2's complement is 0b111111111111 (padding with leading 1's)

So the opcode for rjmp:

1110	kkkk	kkkk	kkkk

would turn into

1110	1111	1111	1111

6.5 Conditional Looping

- Branching is necessary for conditional looping
- Best to think about branching in terms of flow charts
- Separate condition checking action from branching action
- We need status registers for this
- a status register is an 8 bit register that indicates status after operations
- Zero flag is important for now
- result of most recent arithmetic operation is 0, then the zero flag is set to true
- When two equal values are compared it is set to true
- otherwise it is false

- andi can also set zero flag

Loop Counting Down:

```
LDI r16, 0x0A
LOOP: NOP
      DEC r16
      BRNE LOOP ;branches if the zero flag is false
END:
    RJMP END
```

Loop Counting Up:

```
LDI r16, 0x00
LOOP: NOP
      INC r16
      CPI R16, 0x0A ;CPI sets or clears the zero flag as appropriate
      BRNE LOOP
END:
    RJMP END
```

6.6 SREG

- Z: Zero flag set if the result of the previous operation is zero.
- V: Twos complement overflow flag (overflow for twos complement means we don't have enough space to represent the number) Conditions:
  - if the result of addition of two negative numbers is positive there has been overflow.
  - if the result of addition of two positive numbers is negative there has been overflow
  - otherwise no overflow (sum of negative and positive cant overflow).
- S: sign change flag (set when the result of the arithmetic operation has a 1 in the MSB) XOR between twos complement overflow and negative flag.
- C: Carry flag (Unsigned overflow)
- N: Negative flag (Set whenever bit 7 of previous operation is set)

6.7 Branching

- BREQ (branch on equal zero)
- BRNE (branch on not equal zero)

6.8 Reading and Writing from Memory

- Since AVR is harvard when we access memory other than registers we have to refer to it by memory address
- We are accessing data memory, whereas program memory is stored in another location.

6.8.1 Reading and Writing User Data (SRAM)

SRAM is volatile memory

Note: Direct is when we load and store directly using a register and a known address, indirect is when we use pseudo-registers to load and store

Motivation: We need to write to data memory because we have a limited number of registers (which are also considered data memory) so to expand that we have much more memory available to use, the addresses are 16-bit values. Each register still only holds 8 bits.

Direct load and store:

- LDS, STS
- $< addr >$ is 16 bits
- $< addr >$ is also usually defined with the help of the assembler

- destination address must be known at compile time
- LDS and STS work with a data memory byte address ¡addr¿ along with a register.

```
LDS R3, <addr>
STS <addr>, R2
```

Indirect load and store:

- destination address may be computed or constructed at compile time.
- The value stored in X can be either static or dynamic
- in this case X is a pseudo register
- X, Y, Z are pseudo-registers used for reads and writes from SRAM
- They are each the concatenation of two bytes (because we need 16 bits to work with memory)
- X = R27:R26
- Y=R29:R28
- Z = R31:R30
- they are also called pointers
- we can refer to the individual registers associated with a pseudoregister by alias
- i.e.: R26 = XL, R31=ZH
- Note: High byte is always odd numbered register, low byte is always even numbered

```
LD R3, X
ST X, R2
```

Example of Direct vs Indirect Addressing:

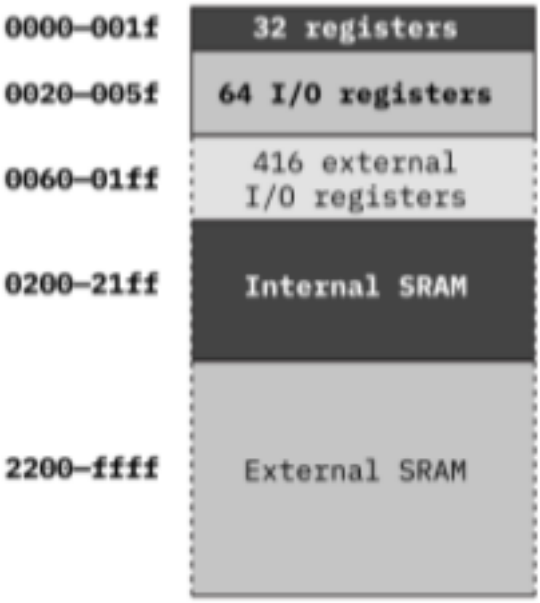
```
LDI r16, 0x77
STS 0x200, r16 ;direct addressing (we know the actual address)


LDI r16, 0x77
LDI r27, HIGH(0x200) ;value: 0x02
LDI R26, LOW(0x200)  ;value: 0x00
ST X, R16 ;indirect addressing (address determined by some operations above)
```

The main different types of addressing

- Direct addressing (reg to reg)
- Direct addressing (literal [constant] to reg)
- Direct data addressing (address of data to reg)
- Indirect data addressing (pseudo register as an address to reg)

6.8.2 Data Memory Map (mega2560)

Data memory $\Longleftrightarrow$ *RAM*



6.8.3 I/O Registers

- I/O registers help us access ports
- Almost every port gives access to eight bits / pins
- Pins can be used for digital input and output
- AVR has 11 ports each of which has three 8-bit registers
  - data direction register (DDRx) (1 = output, 0 = input)
  - pin output register (PORTx)
  - pin input register (PINx)
- Each port has 8 pins
- these pins can either be set to input or output
- If the pins are set to output PORTx register is used to write 0 or 1 to a specific pin
- If pins are configured for input PINx register reads if there is a 0 or a 1 on the pin.

6.8.4 Channels vs Data Addresses

- Channels are faster (addresses must be known at compile time) use IN OUT ops. (only ports A-H use channels).
- Data memory addresses allow port-register addresses to be determined at run time.(Ports A through L use data memory addresses).

6.9 Assembly Language

6.9.1 Background

- Assembly language and assembler are just tools to automate encodings
- Machine instructions are turned into binary.
- Assembler = macro processor and bookkeeper
- Machine instructions are also known as mnemonics
- Two files are created during assembly: .hex and .eep

- these contain bytes generated by assembler
- Code is translated in two passes
- One: Define symbols and labels using location counters
- Two: Convert instructions and data into bit sequences with correct addresses.
- Assembler is case insensitive.

6.9.2 Directives

- Directives give control to the program
- allow us to define constants, set aside memory for variable data
- organize memory
- Stored in program memory with the code itself (goes into flash which is non volatile)
- .db reserves a byte in program memory
- .dw reserves a word to store data
- they both allow us to embed byte values within the binary (program memory)
- Very different than using the data memory
- we can also use directive to set aside addresses in SRAM
- Define symbolic names for registers
- Create constants (equate symbol values to value or expression)

```
start: .db  13, 18, 21, 30 ;4 bytes with these values
sevens: .db 0x07, 0b0000111, 7; three versions of seven
maxint: .dw 32767 ;max 16 bit twos complement number

buff: .byte 32 ;32 bytes in SRAM at address buff
.def sum=r16 ;register 16 can now be refered to as sum
.undef sum ;sum no longer has a meaning
.equ cr=0x0D ;where 'cr' appears, replace it with 0x0D

.cseg ;everything that follows goes into the code segment (flash storage, non-volatile)
.dseg ;everything that follows goes into the data segment (ram, volatile)
```

- Assembler maintains location counters for .cseg, .dseg, and .eseg
- this means that we can place them wherever we'd like in relation to their layout on the screen
- But when the assembler does its first pass to get everything in order it will correctly place things where they need to go
- we could have:

```
.cseg
.org 4 ;begin assembling at address 4
    add r17, r16
    mov r0, r17
    sts RESULT, r0

.org 0 ;begin assembling at address 0
;this code comes before the .org 4 code in the .hex file (still works perfectly fine)
    ldi r16, 3
    ldi r17, 0
    add r17, r16
    add r17, r16
```

6.10 Subroutines

TODO:

6.11 Analog to Digital Conversion

Main difference between analog and digital ports:
Analog ports are a range of values that can be read $[0, 2^{10} - 1]$
Digital ports: two values, either 0 or 1.

- We set a reference voltage for the analog port. i.e. 5V (standard for arduino)
- means we have range from 0-5 volts broken up over $2^{10} - 1$ different values
- i.e. if we use 4 out of the 5 volts we have $\frac{4}{5} * 2^{10} - 1 =$ value on analog port mapping.
- There is also 8 channels which equates to 8 ports we can use as ADC (Analog to digital conversion) ports
- for reference see the ATMega 2560 data sheet.

6.11.1 Register Initialization

- We need to initialize certain registers before the ADC can digitize analog signal
- ADCSR(A/B) = Control and Status Register A/B
- ADMUX = Multiplexer Selection Register: This selects a channel because the ADC can handle up to 8 channels
- ADC = 10-bit ADC Data register (why 10 bits? because of the 2^{10} values)
- the ADC is therefore split into two registers (ADCL (8 bit) and ADCH (2 bit))

6.11.2 ADCSRA

ADCSRA – ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0	
(0x7A)	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Important bits for us:

- ADEN (bit 7) enables the analog to digital conversion
- ADSC (bit 6), setting this to one will start the conversion from analog to digital. (We need to keep checking this to tell when the ADC conversion is done, it will flip back to 1 when finished).

6.11.3 ADMUX

ADMUX – ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0	
(0x7C)	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0	ADMUX
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

A

Important bits:

- REFS1:REFS0 (bits 7/6) select the voltage reference or the ADC (we use 01 for arduino because AVCC has external capacitor at AREF pin
- ADLAR (bit 5) sets a right or left adjustment (whether the number is padded with leading or following 0s) we use 0 (right adjust).
- MUX4:0 are used as the channel selection bits, (see data sheet for all channels) we use 00000, to select ADC0.

6.12 Subtraction

If we have subtraction of unsigned numbers and the subtrahend (quantity to be subtracted) is bigger than the minuend (quantity to be subtracted from) then because the numbers are unsigned rather than having a negative numbers we have a borrow (underflow) which is reflected in the carry flag.

```
   LDI r17, 70
   LDI r16, 30
   SUB r17, r16
   SUB r17, r16
   SUB r17, r16 ;this final operation will cause the carry flag to be set because of the underflow
```

6.13 32-Bit Addition

```
;we want to compute (M + N) : M = 1073743811, N = 535725793 as decimal
; equivalents in hex
;M = 0x400007C3
;N = 0x1FEE86E1

.equ M = 1073743811
.equ N = 535725793

LDI r16, (M & 0xFF)
LDI r17, ((M >> 8) & 0xFF)
LDI r18, ((M >> 16) & 0xFF)
LDI r19, ((M >> 24) & 0xFF)

LDI r20, (N & 0xFF)
LDI r21, ((N >> 8) & 0xFF)
LDI r22, ((N >> 16) & 0xFF)
LDI r23, ((N >> 24) & 0xFF)

ADD r20, r16
ADC r21, r17
ADC r22, r18
ADC r23, r19
```

- Breakdown of the above
- We are masking off the least significant byte to load into each register
- Through the AND 0b11111111 operation
- Using the symbols >> and & rather than instructions allow us to maintain the value in the constant. The assembler computes what is supposed to be there for us and places it in on assembly.

6.14 Control Flow

- RJMP (program label), relative jump (non conditional)
- BR(condition) (program label) branch, conditional
- there are 20 branch operations in AVR

```
    .cseg
    .def a=r16
    .def b=r17
    .def c=r18
    .def d=r19

    ;(1) checking if a >= b
    ;item but in ASM we check if a < b and branch to skip
    cp a, b
    brlo skip ;branch if low
    inc b
skip:
```

```
    ;(2) checking if a > b
    cp a, b
    brsh skip; skip if a <= b
    dec b
skip:

    ;(3) check if m >= 0
    cp m, n
    brlt skip ;skip if m < n
    sub m, n
skip:


    ;(4) check if m == 0
    tst m
    brne skip
    clr n
skip:
```

1. BRLO: Will branch if the carry flag is set (cp $\iff$ Rd-Rr) but only if the number in register Rd was smaller than the number in Rr. Rd < Rr.
2. BRSH: branch if same or higher. Branches if the carry flag is set, only branches if Rd was >= Rr.
3. BRLT: branch if less than (signed) will branch if the signed flag is set iff the signed number in Rd was less than the signed number in Rr.
4. TST: tests if a register is zero or negative by performing a logical AND between register and itself. Sets Z if register is zero or negative
    - BRNE: Branch if not equal tests the zero flag, if the zero flag is cleared it branches.
    - CLR: clears the register.

6.14.1 RJMP vs JMP

- JMP is an unconditional jump, will jump directly to the given address (4M possible jump addresses. Larger than Arduino to account for other hardware with more addresses.)
- RJMP has value added to the program counter so it jumps relative to where it currently is.
- We have two different ways to jump because RJMP requires only two cycles and two bytes whereas JMP requires four bytes and four cycles.
- RJMP uses less CPU and helps when memory is limited.

6.14.2 PROGRAM MEMORY $\neq$ DATA MEMORY

PROGRAM MEMORY

- Program memory can be up to 4M words in size in AVR architecture
- Actual program memory depends on program counter size (restricted in the addresses it can point to).
- Since the ATMega2560 has a 17-bit program counter (3 byte) it can access up to 128K words

DATA MEMORY

- May be up to 64K bytes in size
- contains a range of 64K addresses the SRAM can use with program data
- the ATMega2560 has 8K bytes in SRAM

6.15 Function Calls

- We can call functions in AVR using *call ¡label¿*
- When we call a function, the function is executed and then the program resumes from the next line after the function. But how does this happen?
- We use the stack

6.15.1 Function Call Semantics

- Before branching with a function call
- The current program counter value is saved onto the stack, which is actually the program memory location following the CALL instruction (as the call has been fetched and decoded and the PC has already been incremented [next program address after the call]).
- The address saved onto stack is called the return address
- After function is complete the ret instruction is executed and overwrites PC with most recent values on stack.
- Call and ret both modify pc
- Stack starts at the highest location in internal SRAM (0x21ff).
- as data is pushed onto the stack it grows downward (toward lower number memory addresses)
- As stack grows heap shrinks
- Since an address is pushed onto the stack in bytes the least significant byte is pushed first, and the most significant byte is pushed last.
- When call is executed these are the exact steps that happen:
  1. Return address gets pushed to the stack
  2. We change the instruction pointer to go to the subroutine (by the label)
  3. The stack pointer gets incremented (always pointing at last free location)
  4. On return the stack is popped and the values are put into the instruction pointer.

6.16 Return Values

- There are certain options for returning values from functions
- A mechanism for returning must be decided by programmers
- Two main mechanisms:
  - Return value stored in GP register (r0-r31)
  - Return value placed on stack

6.16.1 Return Value in GP Register

- Similar to a global variable
- Function is called, register is altered in function
- Result of that change is stored in the register

6.16.2 Return Value on Stack

- Simple approach might seem to be just pushing the return value while still inside the function
- This wont work because we still have the return value on the stack, the return result gets in the way
- Instead we pop return value from the stack, then push to the stack, then return
- This makes it so that our return value is on top of the stack

6.17 Function Parameters

Main Mechanisms

- Parameters placed in registers
- Parameters placed on stack

Difference between call by reference and call by value

- call-by-value: when the value of the argument is copied into the function parameter (in other words the value is passed to the function)

- call-by-reference: the memory location of the argument is passed to the parameter of the function.

Example:

Add3 Call by Value:

```
.cseg
.org 0

;value A r25:r24
ldi r25, HIGH(1210)
ldi r24, LOW(1210)

;value B r23:r22
ldi r23, HIGH(400)
ldi r22, LOW(400)

;value C r20
ldi r20, 30

rcall add_three_V
sts RESULT+1, r25
sts RESULT, r24

stop:
    rjmp stop

add_three_V:
    add r24, r22
    adc r25, r23
    clr r21
    add r24, r20
    adc r25, r21
    ret
.dseg
.org 0x200
RESULT: .byte 2 ;allocates 2 bytes in data memory
                ;starting at 0x200
```

Add 3 Call by Reference

```
;initializing the addresses (memory locations) of the numbers to be added
ldi r25, HIGH(A)
ldi r24, LOW(A)

ldi r23, HIGH(B)
ldi r22, LOW(B)

ldi r21, HIGH(C)
ldi r20, LOW(C)

add_three_R:
    ;copy the address for a into X
    mov r27, r25
    mov r26, r24

    ;We assume that the two consecutive addresses hold the values for A
    ;because A is 16 bit
    ;load the first byte of A
    ld r16, X+
    ;load the second byte of A
    ld r17, X

    ;repeat for B
```

```
    mov r27, r23
    mov r26, r22
    ld r18, X+
    ld r19, X

    add r16, r18
    adc r17, r19

    mov r27, r21
    miv r26, r20
    ld r18, X+
    ld r19, X

    add r16, r18
    adc r17, r19

    mov r27, r25
    mov r26, r24

    ;save the result back in X address
    st X+, r16
    st X, r17
    ret
```

6.18 Stack

- In order to use the stack it needs to be initialized
- Usually stack is placed at the top of RAM (0x21ff)
- Just a convention (but should typically be done this way)

```
;initializing the stack

.cseg
.org 0

ldi r16, LOW(RAMEND)
ldi r17, HIGH(RAMEND)

out SPL, r16
out SPH, r17

;we can also push and pop onto the stack

ldi r16, 0xAA
ldi r17, 0xBB
push r16
push r17
...
pop r16
pop r17

stop:
    rjmp stop
;this code will restore the registers to their previous pushed values
;notice that the pops need to be done in reverse order than the pushes
```

6.18.1 Stack Frame

- Code treats the stack as a block of memory holding parameter data
- Accesses the stack using LDD with an offset.
- Stack region that is specific to the parameters is called the stack frame

- In order to use this pattern the caller can push the parameters onto the stack and then call the function
- The callee must know how many parameters were pushed onto the stack to adjust a copy of the stack pointer to retrieve the parameters.

Example, add3 using stack frame:

```
    .cseg
    .org 0

    ;initialize stack pointer
    ldi r17, HIGH(0x21ff)
    ldi r16, LOW(0x21ff)
    out SPH, r17
    out SPL, r16

    ;push the values for A
    ldi r16, HIGH(1210)
    ldi r17, LOW(1210)
    push r17
    push r16

    ;push the values for B
    ldi r16, HIGH(400)
    ldi r17, LOW(400)
    push r17
    push r16

    ;push the values for C
    ldi r16, 30
    push r16

    rcall add_three_sf

    pop r16
    pop r16
    pop r16
    pop r16
    pop r16
stop:
    rjmp stop
```

- Now the values for A, B and C are on top of the stack ahead of the return address from the rcall to add3
- in order to retrieve them we have to reach backwards on the stack by storing the current stack pointer and reaching back to where the parameters are saved

```
    add_three_sf:
        .set PARAM_OFFSET = 8
        ;saving these values so they dont get modified on return
        push r17
        push r16
        push YH
        push YL

        ;save the current location of the stack pointer to Y
        in YH, SPH
        in YL, SPL
        ;we are not able to do indirect access with SP

        ;reaching back onto the stack to pull values for A
        ldd r25, Y + PARAM_OFFSET + 4
        ldd r24, Y + PARAM_OFFSET + 3
        clr r17
        ldd r16, Y + PARAM_OFFSET + 0
        add r24, r16
```

```
    adc r25, r17

    ldd r17, Y + PARAM_OFFSET + 2
    ldd r16, Y + PARAM_OFFSET + 1
    add r24, r16
    adc r25, r17

    pop YL
    pop YH
    pop r16
    pop r17
```

6.18.2 Local Variables in Stack Frame

- Set aside room for use later for local variables
- Code for setting up the stack frame for use like this is called prologue
- Performed before functionality in function
- The size of the stack is increased to make space for local variables
- Two main instructions we should know: adiw (add immediate word) and sbiw (subtract immediate word)
- Main benefit is that it can recursively keep allocating stack frames and we dont have to worry about overwriting registers
- Fibonacci good example
- Fib recursive works as follows:
  - $f_n = f_{n-1} + f_{n-2}$
  - Subtract 1 from n
  - Find fib(n) and store it in a
  - subtract 1 from n again
  - store it in a different variable b
  - compute and return a+b

```
compute_fib:
    ;prologue
    push YH
    push YL
    in YL, SPL
    in YH, SPH

    ;this is what makes space for the local variables
    sbiw YH:YL, 4

    ;need to subtract 4 bytes because a, b 16 bits
    ;two bytes each

    out SPL, YL
    out SPH, YH
    ;end prologue

    ;main part of fib code
    ;loads r16, with the param value initially pushed on stack
    ;local a: +1 from Y
    ;local b: +3 from Y

    ldd r16, Y+10
    cpi r16, 2
    brlo compute_fib_base_cases

    ldi r16, Y+10
    dec r16
```

```
    std Y+10, r16
    push r16
    call compute_fib ;when we recursively call compute fib
                     ; stack frames are already allocated for recursive call
                     ;done in prologue
    pop r16
    std Y+1, r24 ;store the previous return value
    std Y+2, r25 ;assists with the recursion

    ;now we just compute b
    ;b = compute_fib(n-1)
    ldd r16, Y+10
    dec r16
    std Y+10, r16
    push r16
    call compute_fib
    pop r16
    std Y+3, r24
    std Y+4, r25

    ldd r24, Y+3
    ldd r25, Y+4
    ldd r16, Y+1
    ldd r17, Y+2
    ;return value for the function
    add r24, r16
    adc r25, r15

    rjmp compute_fib_epilogue

    compute_fib_base_cases:
        ldi r24, LOW(1)
        ldi r25, HIGH(1)

    ;main part of compute fib ends

    ;tears down the stack frame
    ;decreases the stack size
    ;restore any pushed registers saved in prologue (pop)
    compute_fib_epilogue:
        adiw YH:YL, 4
        out SPL, YL
        out SPH, YH
        pop YL
        pop YH
        ret
```

7 Interrupts and Timers

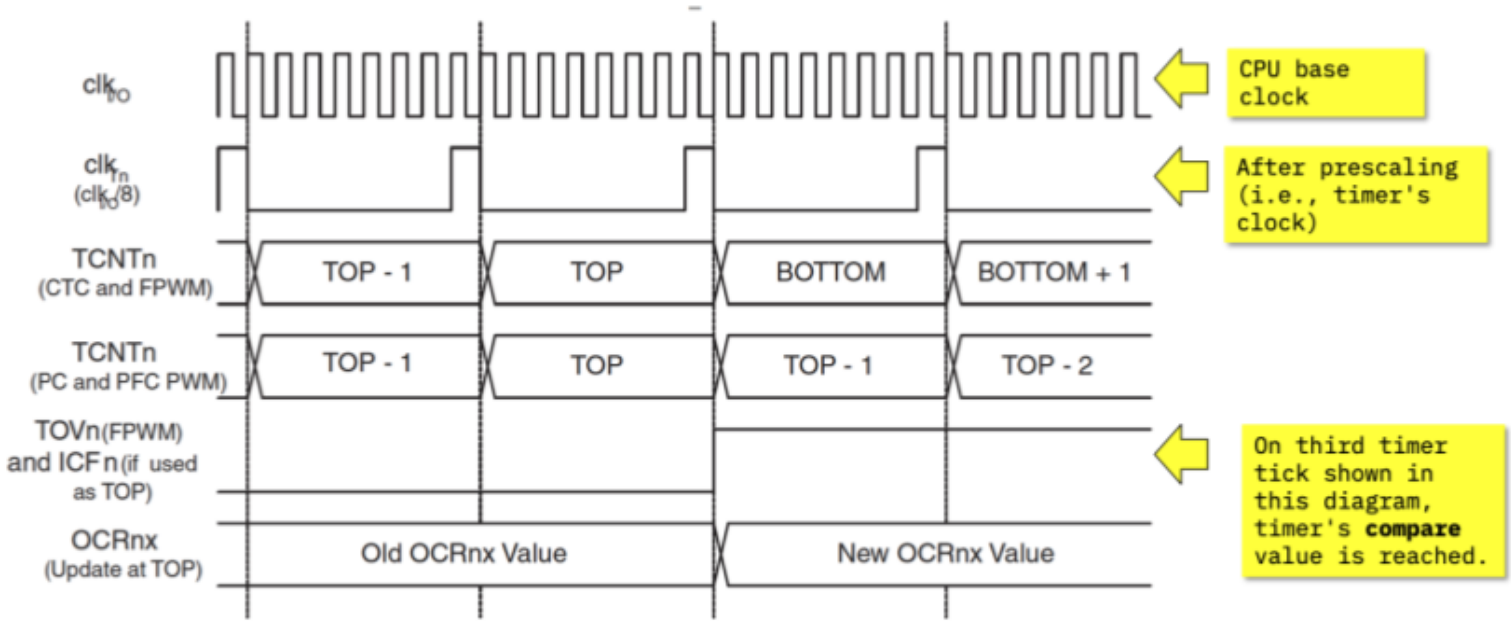
- Timers are just a device that keeps track of time between two events
- Counts a number of clock pulses and signals the CPU when the count is reached
- AT2560 has many timers (8bit: 0,2) (16bit: 1,3,4,5)
- Timers act independently of the ALU
- Depend on the CPU clock for their source of oscillations
- They tick independent of code written by programmer

7.1 Vocab

- Prescalar: number used to slow down the clock

- TCNT: Timer counter register
- OCR: Output compare registers
- TCCR: Timer/Counter control register
  - We specify a timer and either A or B
  - Timer control register is register pairs?
- OC: Output Compare register
- Polling: The state of some external device is checked by the CPU via an infinite loop.

Diagram of ticks after scaling:



7.2 Polling:

Positives and negatives:

- Polling is straightforward
- We can just add more else if clauses when we add more devices to poll
- CPU is always busy
- Mixing polling with non-polling code can be hard.

7.3 Interrupts

- An interrupt is an event that can halt (temporarily) the control flow of the current program
- Events can be triggered by: external (CPU/IO devices), internal devices (CPU timers, CPU errors)
- Running program is temporarily suspended while the task is completed
- CPU takes care of transfer of control to interrupt handler and back
- Interrupted code does not know that interrupt has occurred (because of CPU handling)
- Interrupt handler needs to use reti (rather than ret) to explicitly cause control to return to interrupted code
- We need to disable interrupts while configuring them because having an interrupt go off in the middle of configuration could be bad.
- After the configuration interrupts are reenabled
- After an interrupting event:
  1. CPU pauses where we are in the code

2. Transfers control to the correct handler for that event
3. the handler does its work
4. handler reenables interrupts
5. finally causes CPU to return control to where we originally paused

7.3.1 Programming for Interrupts

- Interrupt handler is always necessary to handle interrupt
- Interrupts are usually defined by the CPU architecture
- ATMega2560 has 57 interrupts including the RESET interrupt
- Enable interrupt flag in order to allow interrupt to be called
- Global interrupts should only be enabled after all devices have been steup
- Note: We should always push SREG to prevent changes happening during execution of interrupts.
- there are two levels to interrupts
  1. The device level interrupt which we must turn on to "turn on the device" such that the device actually sets an interrupt flag.
  2. The global interrupts such that the CPU responds to interrupt flags being thrown. (sei: enables interrupts) (cli: disables interrupts)

7.3.2 Interrupt Vectors

- Special table that associates interrupts with interrupt handler
- Usually, vectors are stored at specific addresses in memory where the interrupt causes the program counter to jump to during an interrupt, then from there we can branch the program flow to the interrupt handler.
- If we dont program the interrupt vector table correctly the assembler will handle the interrupt with a possibly random address.

7.3.3 Interrupt Handlers

- Interrupt handlers should be the smallest amount of code necessary
- Interrupt handlers should rarely call other functions

7.3.4 Non Maskable Interrupts

- NMI: RESET signal is never able to be ignored
- this is known as a non maskable interrupt (NMI)

7.3.5 Multi Level Interrupts

- ATmega only supports single level interrupts
- Some computers support multi level interrupts
- Some interrutps have higher priorities.

8 Performance Issues

8.1 Pipelining

- An instruction execution cycle can run sequentially, where before fetching the next instruction it waits for the current instruction to execute.
- Instructions can be pipelined where during the decode of the previous instruction the next instruction is already being fetched.
- Cycle time is shrunk (clock frequency increases)
- Each phase of the instruction fits within a single clock cycle, for different instructions (i.e. during cycle 8:
  - Instruction 5 is being write back

  - Instruction 6 is being executed
  - Instruction 7 is being decoded
  - Instruction 8 is being fetched
- Overclocking a CPU also increases speed but at the cost of heat.
- in order to achieve this registers are placed between each stage of the pipeline
- Allows us to read from future or previous registers
- Pipelining Hazards:
  1. Control Hazard (branching issues)
  2. Structural Hazard (more than one instruction in the pipeline needs the same CPU resource in the same cycle
  3. Data Hazard (instructions in earlier stages require data that will be result of instruction in later stage)

8.1.1 Branching Issues

- If a branch happens then instructions still in the pipeline are discarded (the work done for them)
- Otherwise they would execute whereas we want the branched instructions to execute.
- To solve this branch prediction is used
- Looks ahead into the program to see if future branches will be taken, starts pipelining instructions for that branch.

8.2 Superscalar Execution

- Multiple instructions are run on each clock cycle (different from pipelining)
- Similar to multiple parallel pipelines

8.3 Data Flow Analysis

- Processor determines which ordering of instructions is the best
- The ultimate goal is to choose an order that executes the fastest

8.4 Speculative Execution

- branch prediction
- data flow analysis combo
- mainline and speculative code can be evaluated in parallel
- This causes vulnerabilities called spectre and meltdown (pretty cool).

8.5 Multicore

- Multiple cores in one CPU
- Different tasks can be executed on different CPU cores
- This allows for real parallel execution
- Multithreaded code possibility
- Threads are completely independent until the output needs to be merged into one result
- also good for having multiple programs run at the same time.

8.6 Hyperthreading

- Make one core look like more than one
- Intel proprietary
- Nothing is done by the software or the developer, happens automatically.

8.7 Single vs Multithreaded

- Single threaded uses one register and one stack
- Whereas multi-threaded is able to use multiple register groups and multiple stacks.

8.8 Speedup Analysis:

- Given some program P
- It takes time T to execute on a single core
- What is the greatest speedup possible when using N processors/comes
- Assume: we have programs which some fraction fo the code can be put into parallel form
- Call this fraction f such that $0 \leq f \leq 1$
- Therefore the fraction of code which cannot be parallelized is (1-f).
- Therefore the benefit from multiple cores comes from:
- For a single core time to execute = T
- If the whole program is parallelizable (if f = 1.0) and we have N cores the new time to execute is $\frac{T}{N}$ (divide the work over N cores)
- If part of the program is parallelizable ($0 < f < 1.0$) and we have N cores, the new time to execute is $T(1-f) + \frac{Tf}{N}$

$T(1-f)$ = portion of code that is not parallelizable,
$\frac{Tf}{N}$ = Fraction of code that is parallelizable multiplied by speed up.

8.9 Amdahls Law

Ratio of pre-improvement time, vs post improvement time is the speedup of the improvement

$$\frac{T}{T(1-f) + \frac{Tf}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

Note that when f is small speedup is negligible
As $N \to \infty$ speedup is bound by $\frac{1}{(1-f)}$

8.10 Measures of Computer Performance

Note: we reuse f but this time as frequency rather than fraction of parallelizable code.

- Processor operations driven by clock which runs at frequency f
- Time between clock pulses: $\tau = \frac{1}{f}$
- Each instruction requires a fixed number of cycles.
- Different types of instruction require different numbers of cycles.

Given a specific program and architecture:

- We can count the instructions I_C (these are the number of machine instructions executed for the whole program until it runs to completion)
- I_i is the count of instructions executed of type i
- CPI_i is the number of cycles needed to executed an instruction of type i
- Overall CPI (cycles to execute an instruction of type i)

We can think of this as the weighted average of instruction types multiplied by their corresponding cycle count.

$$CPI = \frac{\sum_{i=1}^{n}(CPI_i \times I_i)}{I_C}$$

The time to execute a given program is:

$$T = I_C \times CPI \times \tau$$

This makes sense because it is pretty much the sum of all of the instruction types and their cycle counts divided by clock frequency.

Example:

- Suppose we have four instruction classes:
- ALU
- Memory (load and store)
- Branches (conditionals)
- Jumps (unconditional)

$$CPI = \frac{(CPI_{ALU} \times I_{ALU} + (CPI_{MEM} \times I_{MEM}) + (CPI_{BR} \times I_{BR}) + (CPI_j \times I_j)}{I_C}$$

9 Formulas to Know:

Single Core time to execute: T
Fraction of parallelizable code: f
Speedup if whole code is parallelizable: $\frac{T}{N}$

Speedup for fractional parallelizable code: $T(1-f) + \frac{Tf}{N}$

Amdahls Law:

$$\frac{1}{(1-f) + \frac{f}{N}}$$

10 Memory

10.1 Memory Types

- volatile (gets deleted on power off)
  - Ram:
  - Dynamic
  - Static
- Non-volatile
  - ROM, Disk, Flash Memory

10.2 Dynamic Memory

- Dynamic memory uses dynamic electrical elements such as inductors or capacitors
- The time constant $\tau = RC = \frac{L}{R}$
- This is the time it takes to fully charge the capacitor (a charged capacitor represents a set bit whereas an uncharged capacitor represents an unset bit).
- R is the resistance
- C is the capacitance
- L is the latency?
- Important Part: each capacitor stores a bit this type of storage is good because we can fit lots of capacitors in the same space (also cheaper), a little slower than static

10.3 Static Memory

- Each bit is stored in a D-FlipFlop
- Each D-FlipFlop consists of 4 NAN gates
- Each NAN gate consists of 6 transistors
- This makes it harder to fit more in the same amount of space.

10.4 Memory Hierarchy

- Constraint a computer system designer should consider:
- How much memory of a certain type can be obtained (capacity?)
- How fast is that memory (access time)?
- How expensive is it (cost per bit)?
- Trade offs:
- faster access time: faster cost per bit
- greater capacity: smaller cost per bit
- Greater capacity: slower access time

10.5 All the Types of Memory

(in order of speed vs cost)

- Register
- Cache
- Main Memory
- Solid State Disk
- Magnetic Disk
- CD-ROM
- CD-RW
- DVD-RW
- DVD-Ram
- Blu-Ray
- Magnetic Tape

We mix and match between this hierarchy in order to have a good combo of speed and memory

10.6 Memory System Characteristics

1. Location
2. Capacity
3. Unit of Transfer
4. Access Method
5. Performance
6. Physical Type
7. Physical Characteristics
8. Organization

10.7 Location

- Is the memory internal or external
- Main memory is internal: usually refers more to the address/data buses rather than actual location
- External storage, usually peripheral, (disk, SSD, tape) usually accessed via I/O controllers

10.8 Capacity

- Number of words available in the memory device
- Number of bytes
- Words have different definitions: AVR words are two bytes
- Other architechtures a word is 4 bytes

10.9 Unit of Transfer

- Determined by the number of electrical lines leading into and out of memory module
- Addressable Units: byte? word? bigger?
- For internal memory the unit of transfer is the number of bits read/written into memory at a time
- For external memory data is transferred in units larger than a word (sometimes called blocks)

10.10 Access Method

- Sequential Access: access made in a linear sequence (like tape) consequencs of physical medium
- Random access: Each memory location can be accessed independent of others at no extra cost
- Direct: Location of memory block determined by physical location
- Associative: Data value retrieved based on the contents of some key to which the value is mapped

10.11 Performance

- Three parameters are used: access time, memory cycle time, transfer rate
- Access Time:
  - For random access: duration of time from the instant an address is presented to the instant the data is available for us
  - For other memory types: time to position read/write head at needed location
- Memory Cycle Time:
  - Time for electrical transients to die out on signal lines
  - Exclusively concerned with system bus
- Transfer Rate
  - Rate at which memory can be transferred into or out of a memory unit
  - $T_n = T_A + \frac{n}{R}$
  - T_n : average time to read or write n bits
  - T_A: average access time
  - n: Number of bits
  - R transfer rate in bits per second
  - Ex for hard drive: T_A = time to position arm over correct track, rotational latency for sector to arrive under read/write head. R = Number of bytes transferred per second with read/write head in position ()

10.12 Physical Type

- Most commmon today:
- Semiconductor memory (SRAM, DRAM, flash, etc)
- magnetic surface memory (HDD)
- tape, optical and magneto-optical

10.13 Physical Characteristics

- Volatile Memory: Information that decays naturally or is lost when power is removed
- Non-Volatile Memory: once recorded, information remains without deterioration (but it may deteriorate over time)
- Semiconductor memory: non-erasable: read only memory (ROM) rewritable EEPROM (UV sensitive)

10.14 Organization

- How are bits physically arranged to form words?
- Banks
- Ranks
- Pages
- NAND
- NOR
- Bits in addition to data bits (error detection, error correction?)

11 C Programming

11.1 Programming ATMega in C

```
#define F_CPU 16000000UL //defines the number of MHz for the CPU
                         //UL is for unsigned long
#include <avr/io.h>
#include <util/delay.h>

int main(){
 DDRB = 0xff;
 DDRL = 0xff; //predefined knows to load registers

 for(;;){
     PORTL = 0b10101010;
     PORTB = 0b00001010;
     _delay_ms(500);
     PORTL = 0b00000000;
     PORTB = 0b00000000;
     _delay_ms(500); //included function in the delay.h file
 }
 return 0;
}
```

11.2 C to Assembly

- Because assembly doesnt have types, C abstracts new types on top of it
- Similar to how we use two registers to store a 16 bit value, C uses many registers (or data locations) chained together in order to create things like int (typically 2 bytes = 16 bits)
- Arrays are just contiguous regions of memory storing values of the same type
- The compiler must calculate the addresses to go to in order to reach our data.

11.3 Bitwise Operators

```
<< n //lsl by n bits
>> n  //lsr by n bits

| //bitwise or
& //bitwise and
^ //bitwise xor
~ //bitwise not
```

```
//for each of these we can also combine assignment
|=
```