Lecture #5
TECH 3233
ver 2.0

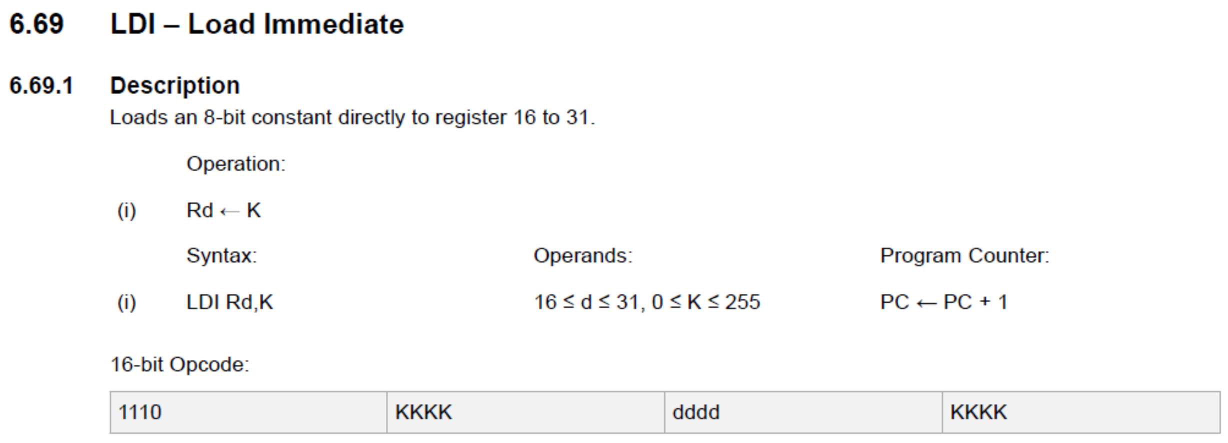
Hand compiling is the process that take an assembly program (using OP Codes) and converts them into the values stored in memory.

Lets take the program:

```
; InClass1.asm
;
; Created: 2/6/2020 7:50:40 AM
; Author : dekohn
;

start:

        LDI     R20, 5
        LDI     R21, 2
        ADD     R20, R21
        ADD     R20, R21
        STS     0x120, R20

L1:     JMP     L1
```

The lines starting with ; are comments. Start: and L1 are labels (used by the compiler for loops and if, so we will start with LDI R20,5.

Looking at the AVR Instruction Set Manual (Available from the class website). Looking up the information on the LDI instruction, we see the following:



6.69 LDI – Load Immediate

6.69.1 Description

Loads an 8-bit constant directly to register 16 to 31.

Operation:

(i) Rd ← K

	Syntax:	Operands:	Program Counter:
(i)	LDI Rd,K	$16 \le d \le 31$, $0 \le K \le 255$	PC ← PC + 1

16-bit Opcode:

1110	KKKK	dddd	KKKK

Since it shows, LDI Rd,K and our instruction is LDI R20, 5……that means we replace d with 20 (dec) and K gets replaced with 5 (dec).

Looking at the 16-bit opcode we now fill in the K's and the d's. So K is 5 (which is also 0x05 hex), so the first group of K's (Most Significant Nibble (MSN)) gets filled in with zeros and the 2nd group (Least Significant Nibble (LSN)). Since the op code sheet shows d is in the range of 16 ≤ d ≤ 31 and we only have 4 bits for d….we need to subtract 16 from 20 to get the value of 4…which we convert to binary and put into d.

So we end up with:

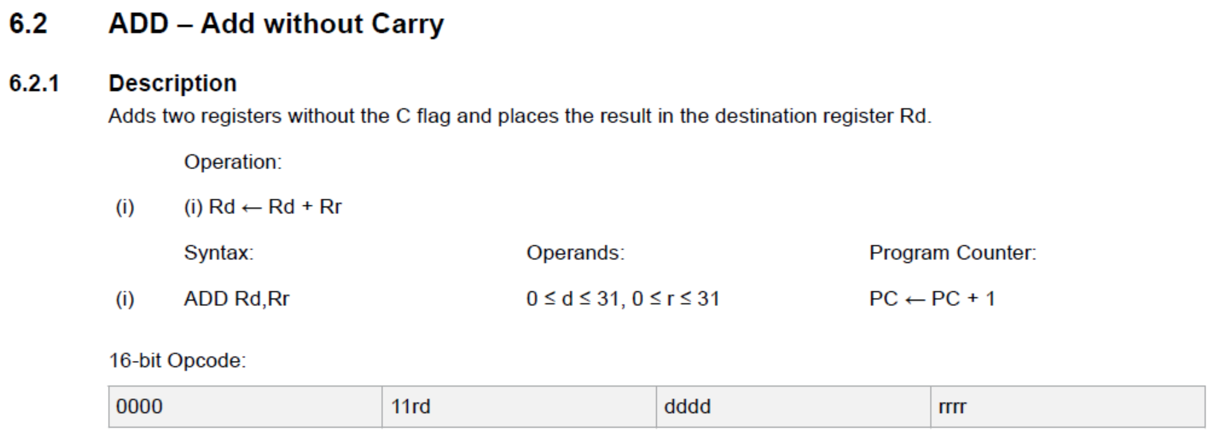
From datasheet	1110	KKKK	dddd	KKKK
Binary	1110	0000	0100	0101
Hex	E	0	4	5

So for that instruction 0xE045 goes into memory.

For the next instruction LDI R21, 2 we follow the same process (0x02 goes into K's and 21-16=5 goes into d's)

From datasheet	1110	KKKK	dddd	KKKK
Binary	1110	0000	0101	0010
Hex	E	0	5	2

For the ADD instructions we once again look up the Op Code in the AVR Instruction Set Manual



6.2 ADD – Add without Carry

6.2.1 Description

Adds two registers without the C flag and places the result in the destination register Rd.

Operation:

(i) (i) Rd ← Rd + Rr

	Syntax:	Operands:	Program Counter:
(i)	ADD Rd,Rr	0 ≤ d ≤ 31, 0 ≤ r ≤ 31	PC ← PC + 1

16-bit Opcode:

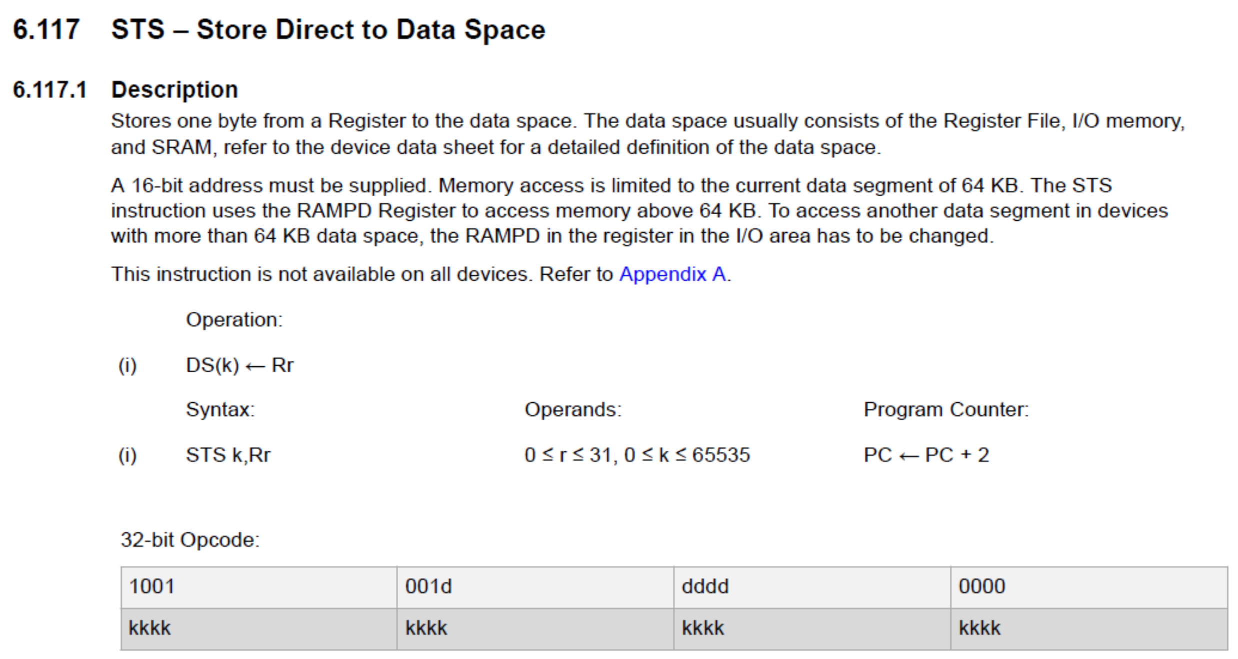
0000	11rd	dddd	rrrr

For the instruction ADD R20, R21 with the OP Code info above, d will be replaced by 20dec and r will be replaced by 21 dec. Note the range for d and r is 0 ≤ d ≤ 31 and 0 ≤ r ≤ 31, so we need 5 bits to represent the full register values….hence why there is an r and d by themselves in one of the nibbles (as seen below). This would be the Most Significant Bit (MSB) of the register number (2^4 or 16 bit). So now our table looks like:

From datasheet	0000	11rd	dddd	rrrr
Binary	0000	1111	0100	0101
Hex	0	F	4	5

For the next instruction, ADD R20, R21 it is the same exact hex value.

Next the STS 0x120, R20 instruction is done. Looking at the Op Code Sheet:



6.117 STS – Store Direct to Data Space

6.117.1 Description

Stores one byte from a Register to the data space. The data space usually consists of the Register File, I/O memory, and SRAM, refer to the device data sheet for a detailed definition of the data space.

A 16-bit address must be supplied. Memory access is limited to the current data segment of 64 KB. The STS instruction uses the RAMPD Register to access memory above 64 KB. To access another data segment in devices with more than 64 KB data space, the RAMPD in the register in the I/O area has to be changed.

This instruction is not available on all devices. Refer to Appendix A.

	Operation:		
(i)	DS(k) ← Rr		
	Syntax:	Operands:	Program Counter:
(i)	STS k,Rr	0 ≤ r ≤ 31, 0 ≤ k ≤ 65535	PC ← PC + 2

32-bit Opcode:

1001	001d	dddd	0000
kkkk	kkkk	kkkk	kkkk

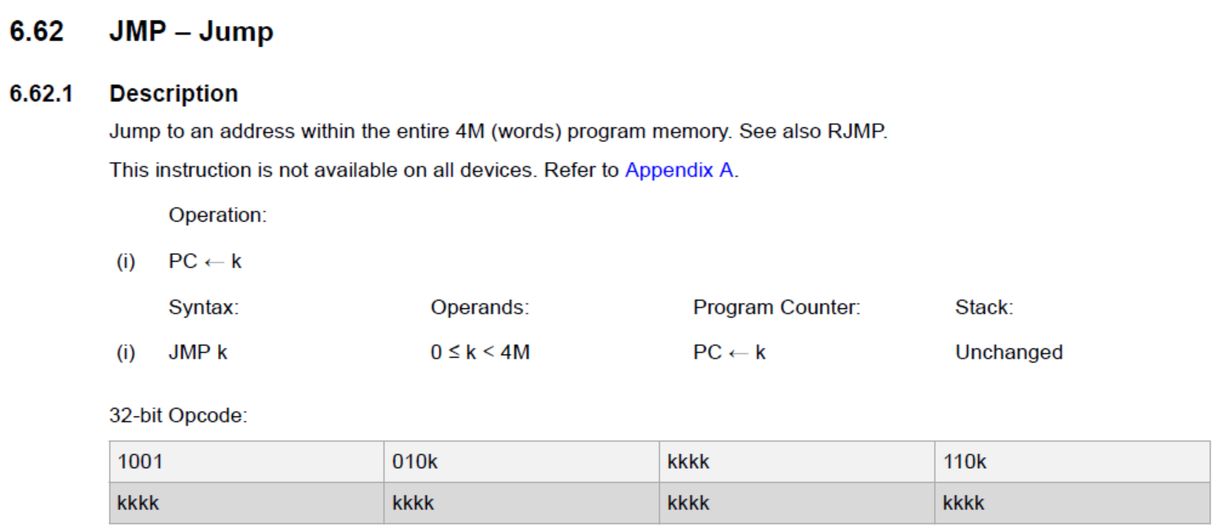
Note that this instruction is 32 bits and k is the full 16bit hex value and the r (shown as d's in the table – probably a misprint) is the register (using 5 bits). So our table would be:

From datasheet	1001	001r	dddd	0000
Binary	1001	0011	0100	0000
Hex	9	3	4	0

From datasheet	kkkk	kkkk	kkkk	kkkk
Binary	0000	0001	0010	0000
Hex	0	1	2	0

So 0x9350 and 0x0120 would go into memory.

The final instruction is the L1: Jmp L1 and the Op Code Sheet is as follows:



6.62 JMP – Jump

6.62.1 Description

Jump to an address within the entire 4M (words) program memory. See also RJMP.

This instruction is not available on all devices. Refer to Appendix A.

Operation:

(i) PC ← k

	Syntax:	Operands:	Program Counter:	Stack:
(i)	JMP k	$0 \le k < 4M$	PC ← k	Unchanged

32-bit Opcode:

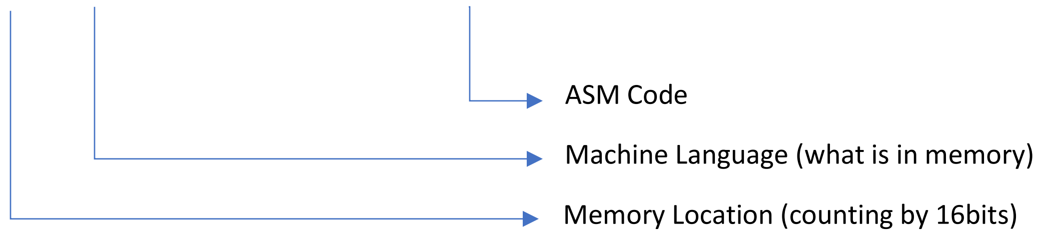
1001	010k	kkkk	110k
kkkk	kkkk	kkkk	kkkk

Again this is a 32 bit instruction, but it is not obvious what values we use as k's.

To understand what value we need, we have to understand what an Assembly Language Program (ASM) does with labels like "L1:". When a compiler runs, it not only does the conversion we have done above, but it also counts the number of memory locations each instruction takes. When it see a label, it marks down the memory location were the instruction it points to is stored, and anywhere it sees the label "L1" used, it replaces it with that value (or in some cases, it uses that to calculate how many memory locations it has to go forward or revers in the code…this is known as a relative jump).

So far we have:

```
000000 e045                        LDI    R20, 5
000001 e052                        LDI    R21, 2
000002 0f45                        ADD    R20, R21
000003 0f45                        ADD    R20, R21
000004 9340 0120                   STS    0x120, R20
```


ASM Code
Machine Language (what is in memory)
Memory Location (counting by 16bits)

So the next memory location would be 000006 (since STS is a 32 bit instruction). Note addresses are 6 nibbles (6 hex characters) and only 0-3 are used for the first nibble (two bits). Hence the two k's that are by themselves.

So for this instruction we would get:

From datasheet	1001	010k	kkkk	110k
Binary	1001	0100	0	1100
Hex	9	4	0	C

From datasheet	kkkk	kkkk	kkkk	kkkk
Binary	0000	0000	0000	0110
Hex	0	0	0	6

So our completed program will be:

```
000000 e045                  LDI    R20, 5
000001 e052                  LDI    R21, 2
000002 0f45                  ADD    R20, R21
000003 0f45                  ADD    R20, R21
000004 9340 0120             STS    0x120, R20

000006 940c 0006         L1: JMP    L1
```

<u>Hand Tracing</u> is the process in which you act as the CPU and do each instruction in order and show how the registers, and memory locations change as the program progresses.

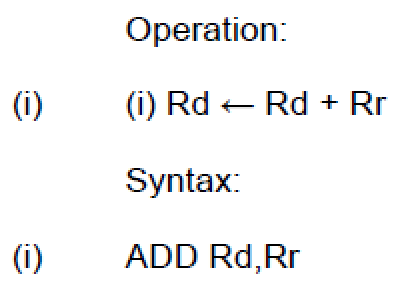
So starting with LDI R20,5 we would have

Reg / Mem	Value
R20	5

Next is the LDI R21,2

Reg / Mem	Value
R20	5
R21	2

Next is ADD R20,R21. Here we note that in the Op Code sheet, we see:



Operation:

(i) (i) $Rd \leftarrow Rd + Rr$

Syntax:

(i) ADD Rd,Rr

This tells us the Rd is added to Rr and then STORED IN Rd, so after the instruction R20 is overwritten by the answer (note we cross off the previously stored value and replace it with the new value:

Reg / Mem	Value
R20	~~5~~ 7
R21	2

The next instruction does the same ADD R20,R21:

Reg / Mem	Value
R20	~~5~~ ~~7~~ 9
R21	2

And lastly the STS 0x120,R20 stores the value in R20 in memory location 0x120

Reg / Mem	Value
R20	~~5~~ ~~7~~ 9
R21	2
0x120	9

.

L1: Jmp L1 has no effect on Registers or the memory location, so the table (above) is the final answer).