18-100 Lecture 19:
Intro to AVR Assembly Programming

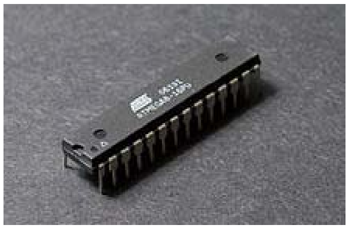
Tom Sullivan
Dept of ECE, CMU
March 29, 2012

Today's Goal: Get ready for Lab 9

Announcements:

Handouts: Atmel 8-bit AVR ATmega8 Databook (on Blackboard)
Atmel 8-bit AVR Instruction Set Manual (on Blackboard)

Your Computer: Atmel ATmega8



[image from Wikipedia]

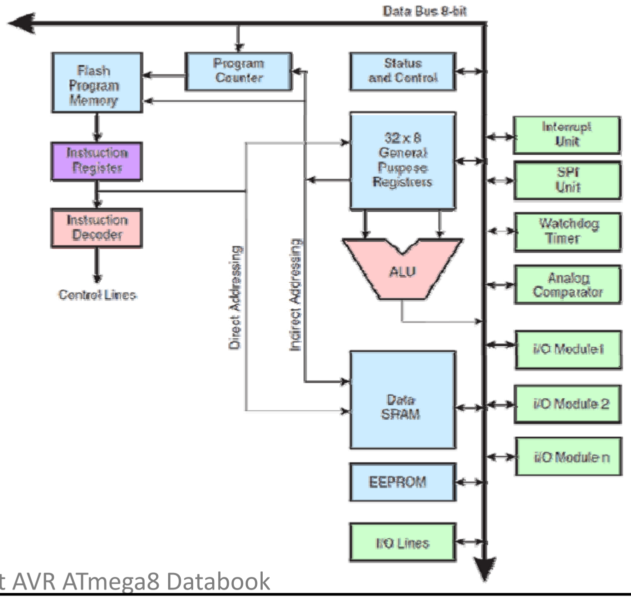
- ~$3.00 each
  - may be ~10K gates
  - clock up to 16MHz
  - 1KB Data SRAM (8-bit words)
  - 8KB Program Memory (Flash)
- BTW, a modern high-end CPU (e.g., Intel Xeon)
  - billions of transistors (10+ cores)
  - many GHz (approaching 100 GFLOPs/sec)
  - 10s of MB in just caches

Atmel ATmega8

state

I/O

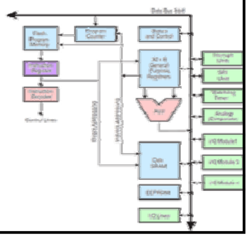
logic



Page 9 Atmel 8-bit AVR ATmega8 Databook

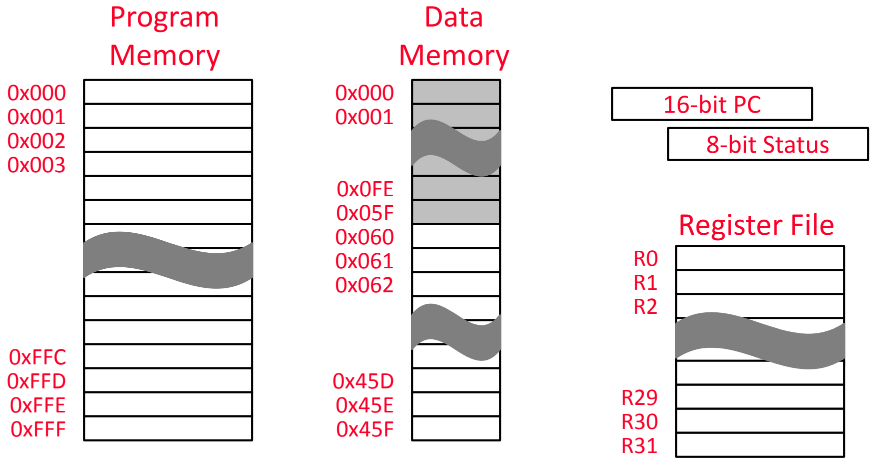
An Instruction Set Architecture

- Abstracting a processor/computer as
  - program visible state
    - memory, registers, program counters, etc.
  - set of instructions to modified state; each prescribes
    - which state elements are read as operands
    - which state elements are updated and to what new values
    - where is the next instruction
- Other details
  - instruction-to-binary encoding
  - data format and size
  - how to interface with the outside world?
  - protection and privileged operations
  - software conventions



"AVR" Program Visible State

(ones we care about for now)

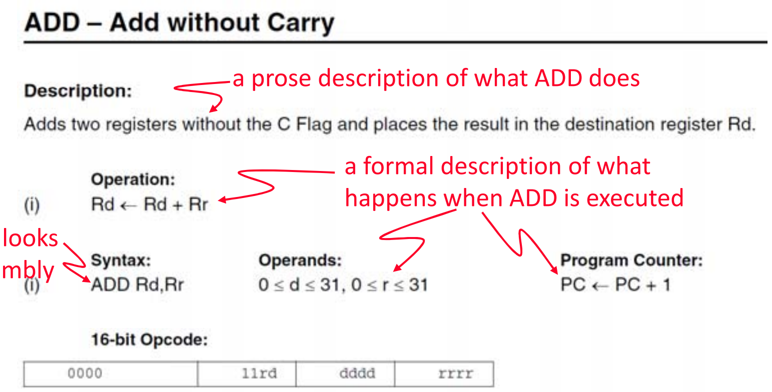


-4K (2^{12}) 16-bit words
-each instruction
either 1 or 2 words

-1K (2^{10}) 8-bit words
-btw 0x060~0x45F
-what's in 0x000~0x05F?

32 8-bit words

AVR Instruction Example: ADD



ADD – Add without Carry

Description: a prose description of what ADD does

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

Operation: a formal description of what happens when ADD is executed

(i) Rd ← Rd + Rr

how it looks in assembly

Syntax: **Operands:** **Program Counter:**

(i) ADD Rd,Rr 0 ≤ d ≤ 31, 0 ≤ r ≤ 31 PC ← PC + 1

16-bit Opcode:

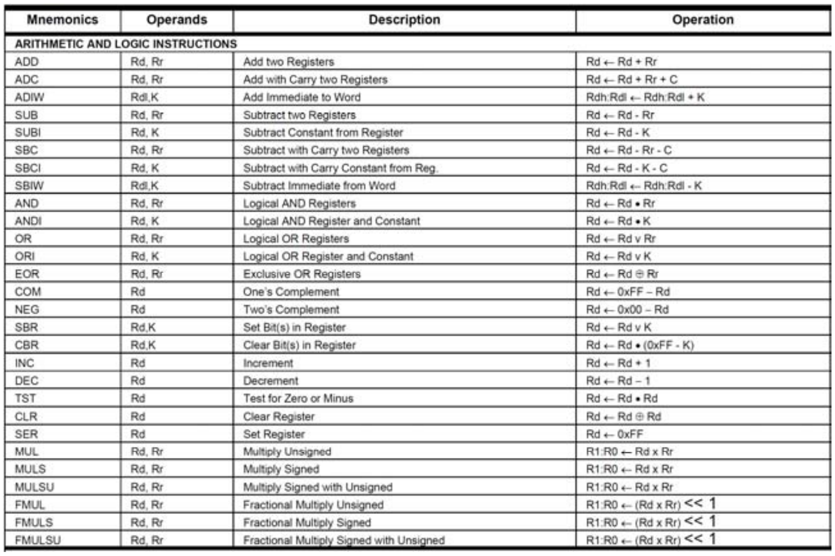
0000	11rd	dddd	rrrr

binary encoding
- "ADD" = 000011 in bit[15:10]
- d=bit[8],bit[7:4]
- r=bit[9],bit[3:0]

Page 17 Atmel 8-bit AVR Instruction Set Manual

Note:
-2 input 1 output function, but Rd is used as both src and dest
- what is this "carry"?

Other ALU Instructions



Mnemonics	Operands	Description	Operation
ARITHMETIC AND LOGIC INSTRUCTIONS			
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr
COM	Rd	One's Complement	Rd ← 0xFF – Rd
NEG	Rd	Two's Complement	Rd ← 0x00 – Rd
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)
INC	Rd	Increment	Rd ← Rd + 1
DEC	Rd	Decrement	Rd ← Rd – 1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd
SER	Rd	Set Register	Rd ← 0xFF
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1

Page 282 Atmel 8-bit AVR ATmega8 Databook

Assembly Programming 101

- Break down high-level program constructs into a sequence of elemental operations

- E.g. High-level Code

```
f = ( g + h ) – ( i + j )
```

- Assembly Code
  - suppose g, h, i, j are in r15, r16, r17, r18 and do not need to be preserved

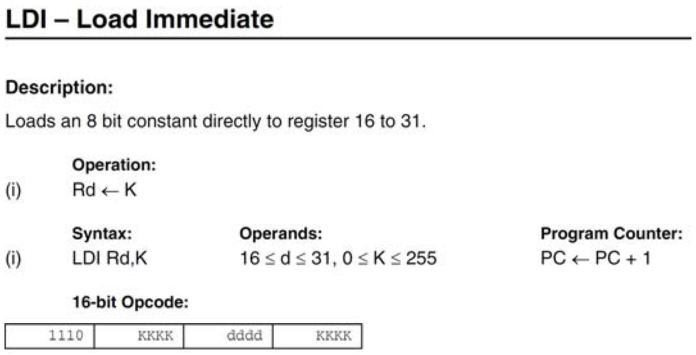
```
add r15, r16     ; r15 = g+h
add r17, r18     ; r17 = i+j
sub r15, r17     ; r15 = f
```

What if we do want to preserve **r15**~**r18**?

General Instruction Classes

- Arithmetic and logical operations
  - fetch operands from specified locations
  - compute a result as a function of the operands
  - store result to a specified location
  - update PC to the next sequential instruction
- Data movement operations
  - fetch operands from specified locations
  - store operand values to specified locations
  - update PC to the next sequential instruction
- Control flow operations
  - fetch operands from specified locations
  - compute a branch condition and a target address
  - if "branch condition is true" then PC ← target address
    else PC ← next seq. instruction

Move "Immediate" to Register



LDI – Load Immediate

Description:

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

Operation:

(i) Rd ← K

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

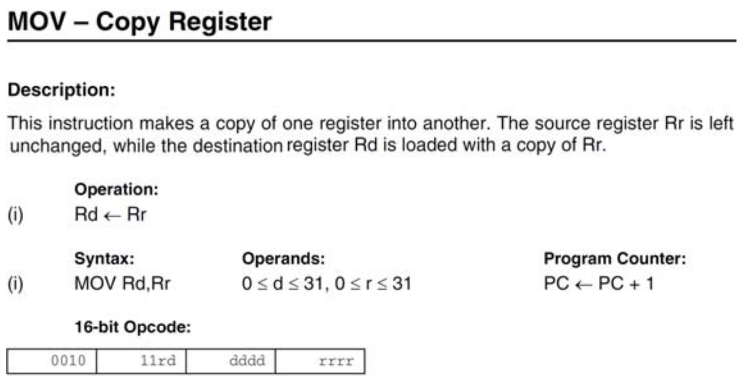
16-bit Opcode:

1110	KKKK	dddd	KKKK

Note:
Rd can only be r16~r31 because in order to give you an 8-bit immediate, there are only 4 bits left to specify Rd

Page 20 Atmel 8-bit AVR Instruction Set Manual

Move Register to Register (Copy)



MOV – Copy Register

Description:

This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr.

Operation:

(i) Rd ← Rr

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

16-bit Opcode:

0010	11rd	dddd	rrrr

We wait until next time to see “load” (i.e., move memory to register) and “store” (i.e., move register to memory)

Page 101 Atmel 8-bit AVR Instruction Set Manual

Assembly Programming 102

- Break down high-level program constructs into a sequence of elemental operations
- E.g. High-level Code

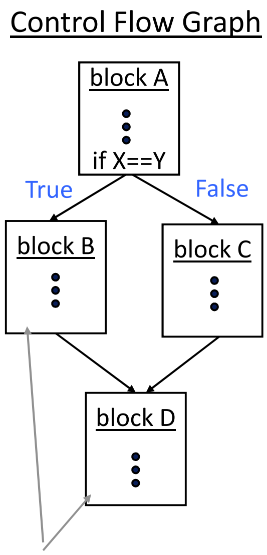
```
f = ( g + h ) – ( i + j )
```

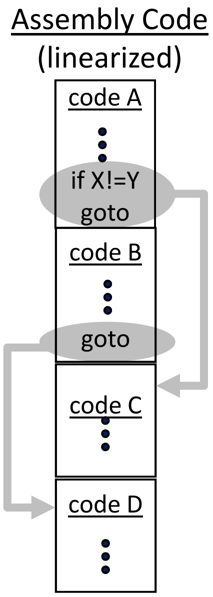
- Assembly Code
  - suppose g, h, i, j are in r15, r16, r17, r18 and should be preserved; put result f in r19; assume r20 is “free”

Control Flow Instructions

- C-Code

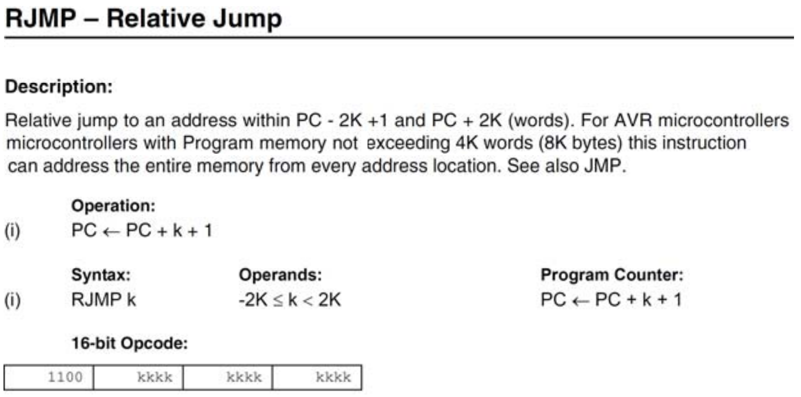
{ code block A }
if X==Y then
 { code block B }
else
 { code block C }
{ code block D }





these things are called basic blocks

Control Flow: Jump!



RJMP – Relative Jump

Description:

Relative jump to an address within PC - 2K +1 and PC + 2K (words). For AVR microcontrollers microcontrollers with Program memory not exceeding 4K words (8K bytes) this instruction can address the entire memory from every address location. See also JMP.

Operation:

(i) PC ← PC + k + 1

	Syntax:	Operands:	Program Counter:
(i)	RJMP k	-2K ≤ k < 2K	PC ← PC + k + 1

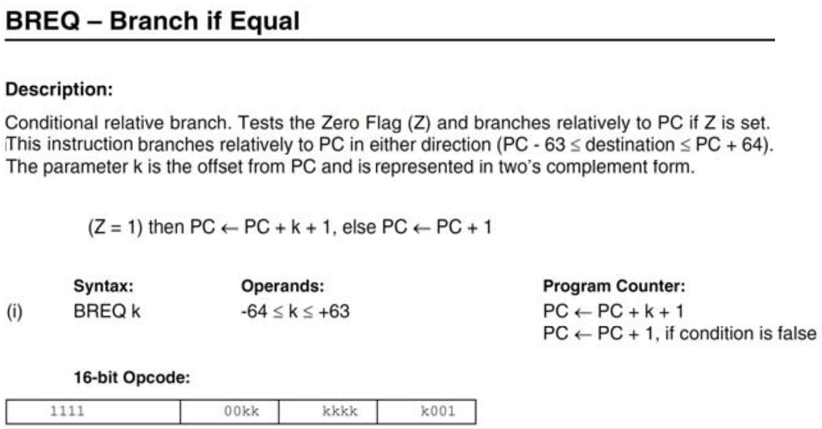
16-bit Opcode:

1100	kkkk	kkkk	kkkk

Note: Jump target is specified as an offset from PC+1, but, fortunately, in assembly programs, you can specify the label of an “absolute” target instruction and the assembler will figure out the offset. (example later)

Page 117 Atmel 8-bit AVR Instruction Set Manual

Control Flow: Branch?



BREQ – Branch if Equal

Description:

Conditional relative branch. Tests the Zero Flag (Z) and branches relatively to PC if Z is set. This instruction branches relatively to PC in either direction (PC - 63 ≤ destination ≤ PC + 64). The parameter k is the offset from PC and is represented in two's complement form.

(Z = 1) then PC ← PC + k + 1, else PC ← PC + 1

	Syntax:	Operands:	Program Counter:
(i)	BREQ k	-64 ≤ k ≤ +63	PC ← PC + k + 1 PC ← PC + 1, if condition is false

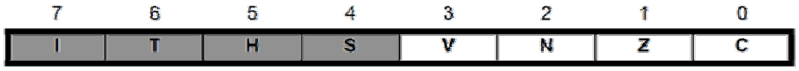
16-bit Opcode:

1111	00kk	kkkk	k001

Note:
- Like in RJMP, a branch target is also PC-relative
- (Z=1) is the branch condition. What the heck is Z?

Page 29 Atmel 8-bit AVR Instruction Set Manual

Status Register



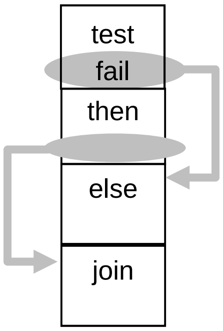
7	6	5	4	3	2	1	0
I	T	H	S	V	N	Z	C

don't worry about 7~4

- 'V', 'N', 'Z', 'C' are arithmetic flags automatically updated after each ALU-class instructions
  - Z: set if the last result was zero,
  - N: set if the last result was negative (2's complement)
  - V: set if the last op caused an overflow (2's comp)
  - C: set if the last op caused a carry (unsigned)
- Each has corresponding branch instructions
  - BREQ/BRNE, BRMI/BRPL, BRVS/BRVC, BRCS/BRCC
- E.g.,
  - after "SUB Rx, Ry" or "CP Rx, Ry", Z is set if Rx==Ry
  - BREQ taken if Rx==Ry, BRNE taken if Rx!=Ry

Page 11 Atmel 8-bit AVR ATmega8 Databook

Assembly Programming 201



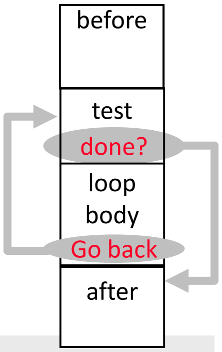
- E.g. High-level Code

```
if (i == j) then
      e = g
else
      e = h
f = e
```

- Assembly Code
  - suppose e, f, g, h, i, j are in r_e, r_f, r_g, r_h, r_i, r_j

```
      cp   r_i, r_j    ; set status flags
      brne L1          ; if i!=j skip to L1 (else)
                       ; assembler computes offset
      mov  r_e, r_g    ; e gets g
      rjmp L2          ; skip to L2 (join)
  L1: mov  r_e, r_h    ; e gets h
  L2: mov  r_f, r_e    ; f gets e
       . . . .
```

Assembly Programming 202



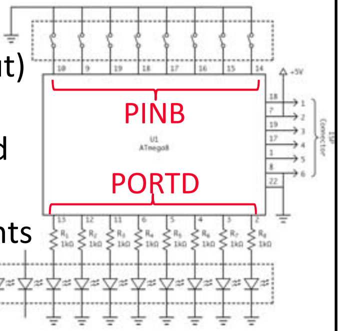
- E.g. High-level Code

```
i=0; j=10;
while (i != j) {
      i++;
}
...
```

- Give it try. Pick your own free registers

I/O

- In Lab 9, you will tie PINB (as input) to dip-switches and PORTD (as output) to LEDs
- The instruction "in Rx, PINB" will load the value at PINB into Rx
- "out PORTD, Rx" will copy the contents of Rx to output PORTD (and hold)



- They are the only I/O operations you need know about
- Please do not feel free to experiment . . . .

When you fiddle with I/O, it is no longer an abstraction. Very real bad things can happen!!

Lab 9 Starter Code

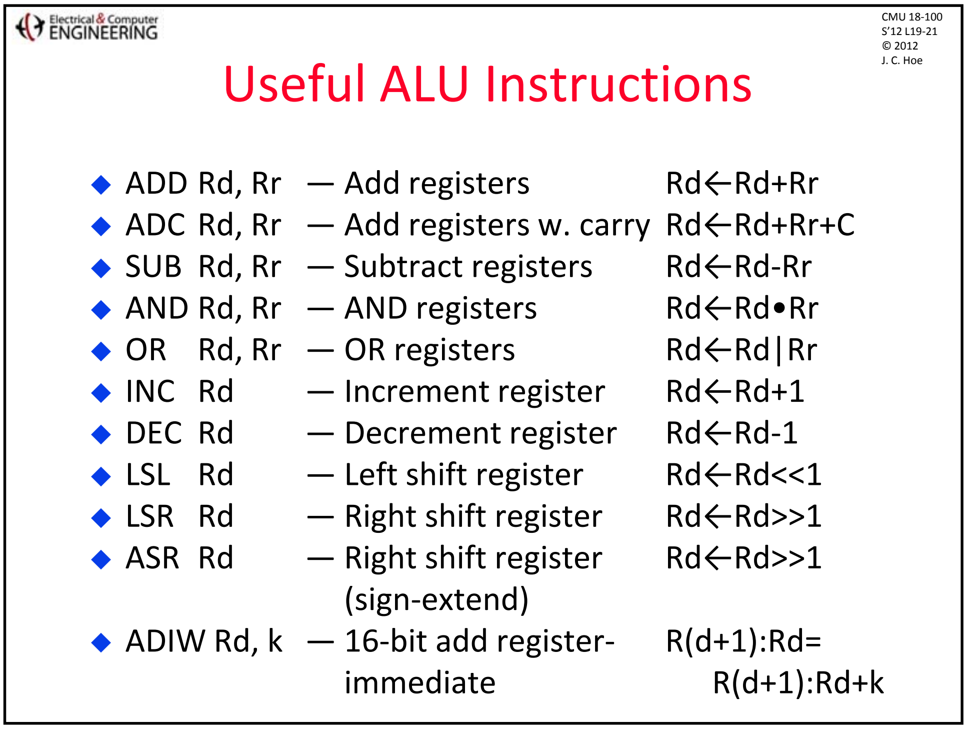
```
.equ PINB=0x03
.equ DDRB=0x04
.equ PORTB=0x05
.equ PIND=0x09
.equ DDRD=0x0a
.equ PORTD=0x0b

.org 0x0000
entry:
        ldi r16,0xFF
        out DDRD,r16
        ldi r16,0x00
        out DDRB,r16
        ldi r16,0xff
        out PORTB,r16
```

DO NOT change the above!!

```
main:
        in r16,PINB       ← read input
        mov r17,r16
        andi r16,0x0f
        lsr r17
        lsr r17           ← compute stuff
        lsr r17
        lsr r17
        add r16,r17
        out PORTD,r16     ← display output
        rjmp main         ← do it again
```

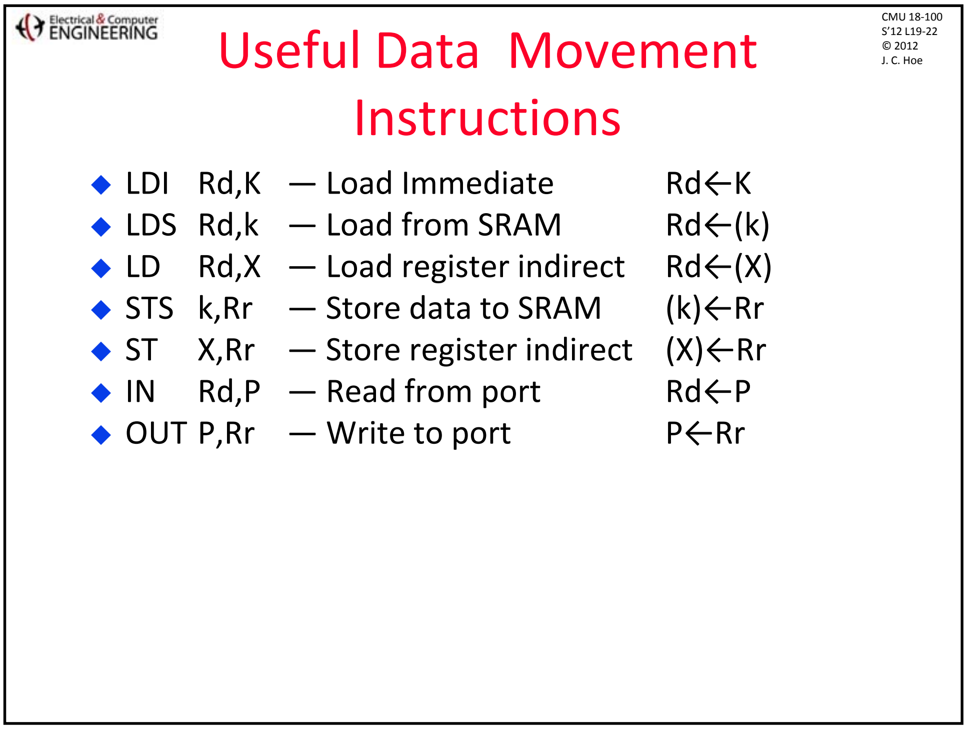
-figure out what the example does
-try it out for real on the board
-modify "compute stuff" to do what Lab 9 asks for
-demo your program on the board



CMU 18-100
S'12 L19-21
© 2012
J. C. Hoe

Useful ALU Instructions

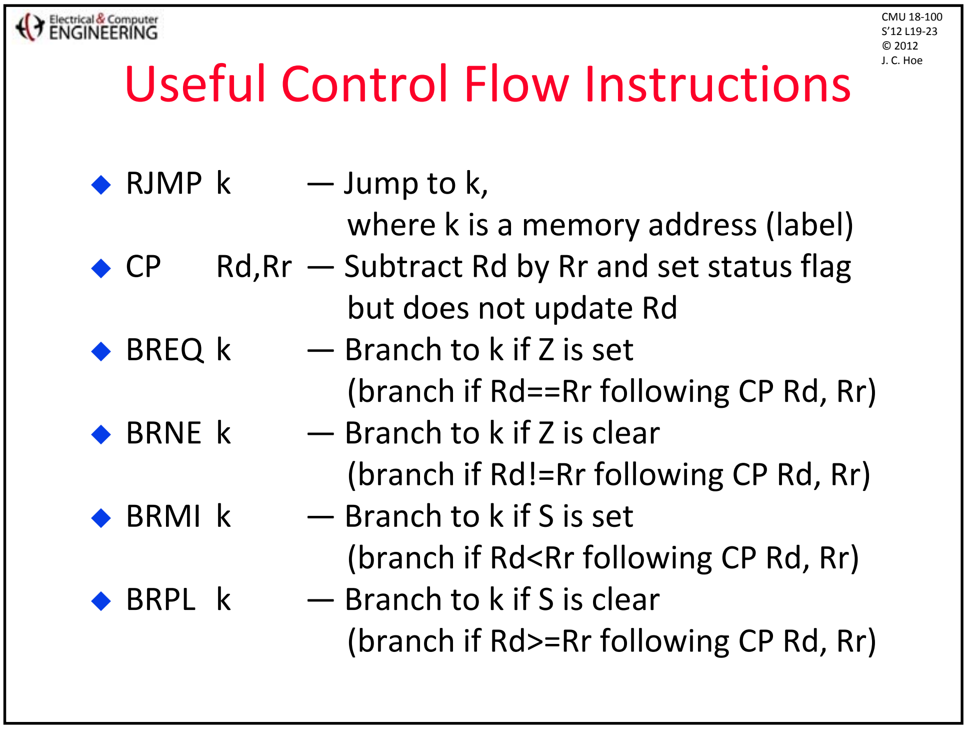
ADD	Rd, Rr	— Add registers	Rd←Rd+Rr
ADC	Rd, Rr	— Add registers w. carry	Rd←Rd+Rr+C
SUB	Rd, Rr	— Subtract registers	Rd←Rd-Rr
AND	Rd, Rr	— AND registers	Rd←Rd•Rr
OR	Rd, Rr	— OR registers	Rd←Rd|Rr
INC	Rd	— Increment register	Rd←Rd+1
DEC	Rd	— Decrement register	Rd←Rd-1
LSL	Rd	— Left shift register	Rd←Rd<<1
LSR	Rd	— Right shift register	Rd←Rd>>1
ASR	Rd	— Right shift register (sign-extend)	Rd←Rd>>1
ADIW	Rd, k	— 16-bit add register-immediate	R(d+1):Rd= R(d+1):Rd+k



CMU 18-100
S'12 L19-22
© 2012
J. C. Hoe

Useful Data Movement Instructions

LDI	Rd,K	— Load Immediate	Rd←K
LDS	Rd,k	— Load from SRAM	Rd←(k)
LD	Rd,X	— Load register indirect	Rd←(X)
STS	k,Rr	— Store data to SRAM	(k)←Rr
ST	X,Rr	— Store register indirect	(X)←Rr
IN	Rd,P	— Read from port	Rd←P
OUT	P,Rr	— Write to port	P←Rr



Useful Control Flow Instructions

- RJMP k — Jump to k,
  where k is a memory address (label)
- CP Rd,Rr — Subtract Rd by Rr and set status flag
  but does not update Rd
- BREQ k — Branch to k if Z is set
  (branch if Rd==Rr following CP Rd, Rr)
- BRNE k — Branch to k if Z is clear
  (branch if Rd!=Rr following CP Rd, Rr)
- BRMI k — Branch to k if S is set
  (branch if Rd<Rr following CP Rd, Rr)
- BRPL k — Branch to k if S is clear
  (branch if Rd>=Rr following CP Rd, Rr)