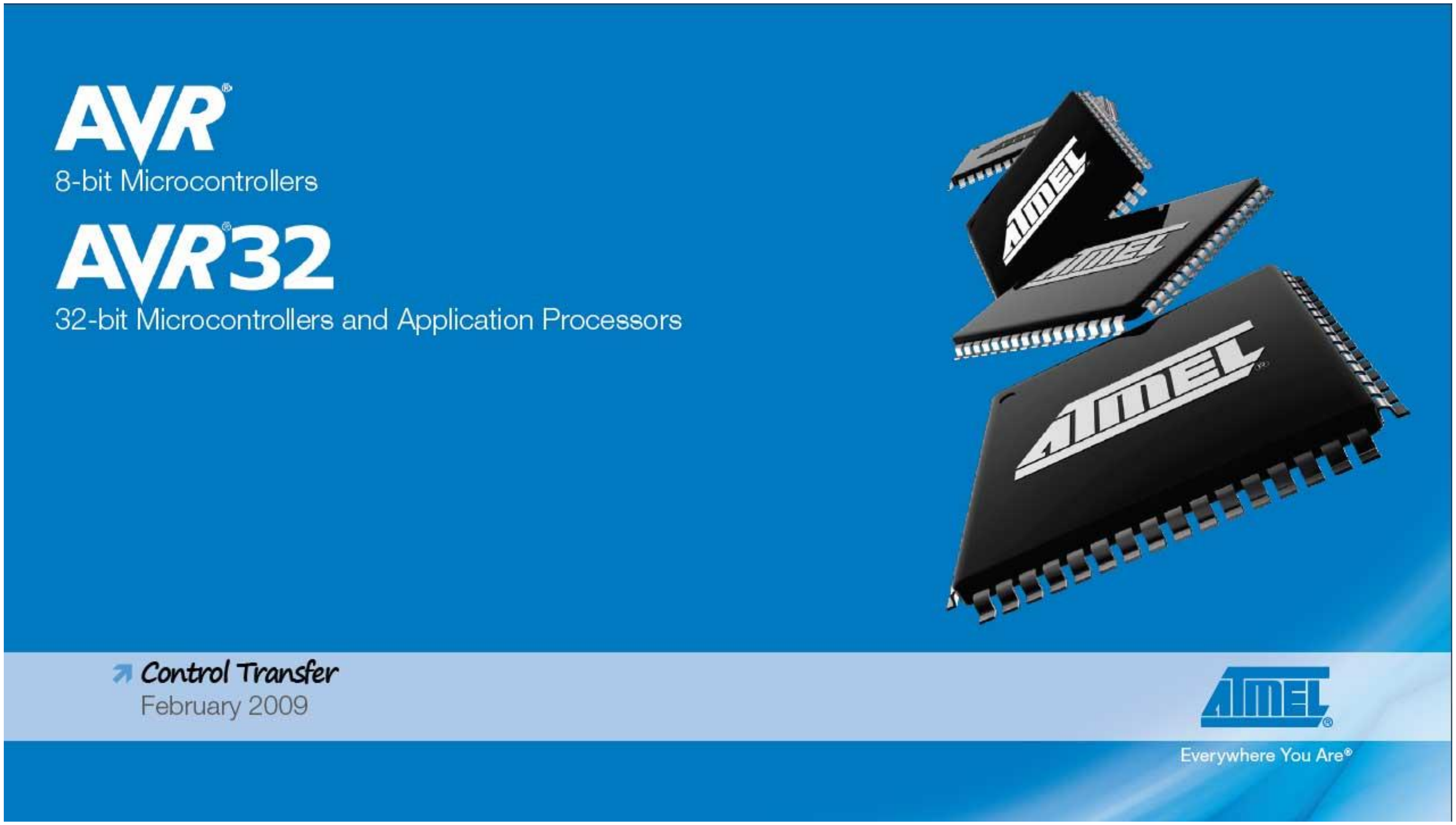


AVR®

8-bit Microcontrollers

AVR®32

32-bit Microcontrollers and Application Processors

Control Transfer

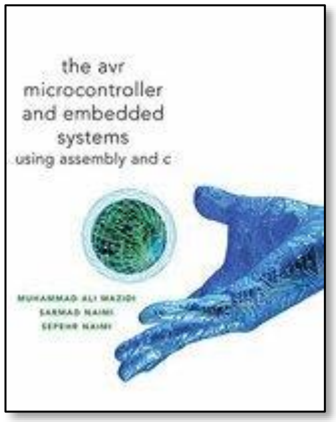
February 2009

Everywhere You Are®

AVR Control Transfer -AVR Looping

READING

The AVR Microcontroller and Embedded Systems using Assembly and C)
by Muhammad Ali Mazidi, Sarmad Naimi, and Sepehr Naimi


the avr microcontroller and embedded systems using assembly and c

MUHAMMAD ALI MAZIDI
SARMAD NAIMI
SEPEHR NAIMI

Chapter 3: Branch, Call, and Time Delay Loop

Section 3.1: Branching and Looping

Section 3.3: AVR Time Delay and Instruction Pipeline

ADDITIONAL READING

Introduction to AVR assembler programming for beginners, controlling sequential execution of the program
http://www.avr-asm-tutorial.net/avr_en/beginner/JUMP.html

AVR Assembler User Guide http://www.atmel.com/dyn/resources/prod documents/doc1022.pdf

Features

[illegible]

8-bit AVR® Microcontroller with 4/8/16/32K Bytes In-System Programmable Flash

ATmega48PA
ATmega88PA
ATmega168PA
ATmega328P

Contents

Loop Example 1: Loop through a block of code 7 times.

- Typically we increment the counter variable in C++.

```
for(int i=0; i<7; i++);  // This statement loops 7 times {i: 0,1,2,3,4,5,6}
```

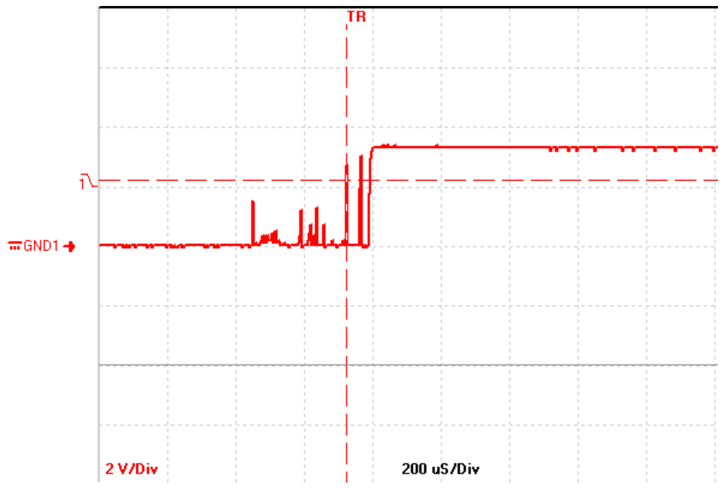
- As shown in the example at the right below, in assembly we decrement the counter variable.
  {i: 7,6,5,4,3,2,1}
  This allows us to immediately test the SREG Z-flag bit without an intermediate compare instruction.

```
C++                                                       Assembly

for(int i=7; i<>0; i--)    int i = 7;                           ldi r16, 7
{                          do                             loop:
Block of code              {                                    Block of code
}                            Block of code                      dec r16
                             i--;                               brne loop
                           } while(i<>0);

1. Initialization          1. Initialization              1. Initialization
2. Test Condition          2. Block of code               2. Block of code
3. Block of code           3. Decrement                   3. Decrement
4. Decrement               4. Test Condition              4. Test Condition
```

Button Debounce Example



[1]

- In the screen capture (red waveform), a button bounces for about 400us when pressed. Once the transition is detected, we want to design a software loop that will do nothing while the switch input stabilizes.
- Specifically, we want to design a software delay routine that will generate a delay of approx. 500 μs.

Delay Calculation for AVR

- We begin by designing a simple loop.

```
wait:
        ldi     r16, _____        // Loop Count
delay:
        dec     r16               //  _____ machine cycles
        brne    delay             //  _____ machine cycles
```

- To discover the delay generated by our "software" loop we begin by finding the answers to the questions.
  - What "Loop Count" L_{cnt} will generate the maximum delay?
  - What is a machine cycle and how many machine cycles are required for each line of code?
  - What is the number of machine cycles N_{mc} in 1 loop?

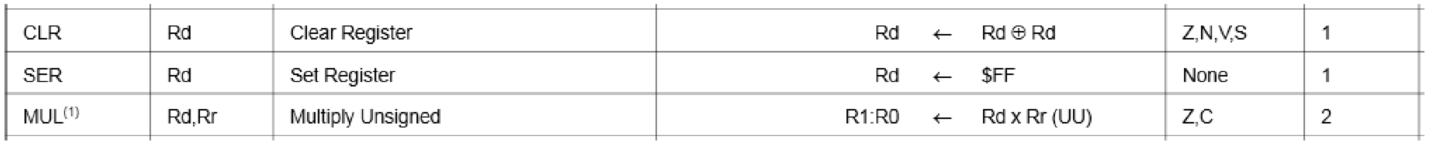
[1] http://generichid.sourceforge.net/buttonbounceDSO.png

Instruction (or Machine) Cycle Time for the AVR

- Machine Cycle – The number of clock cycles it takes the CPU to fetch and execute an instruction.
- Because the AVR processors incorporate a 2-stage pipeline, there is a one-to-one relationship between an AVR machine cycle and a clock cycle. In contrast for the non-pipelined 8051 microcontroller one machine cycle = 12 clock cycles.
- Therefore to calculate the time it takes for one machine cycle you only need to take the inverse of the clock frequency.

$t_{mc} = 1 / f_{clk}$ Example: $f_{clk} = 16\ MHz$ $t_{mc} = 1 / 16\ MHz = 0.0625\ \mu s\ (62.5\ ns)$

- As shown in the "Complete Instruction Set Summary" on page 11 of the AVR Instruction Set Document (Atmel doc0856) most AVR instructions need only one or two clock cycles to fetch and execute an instruction.

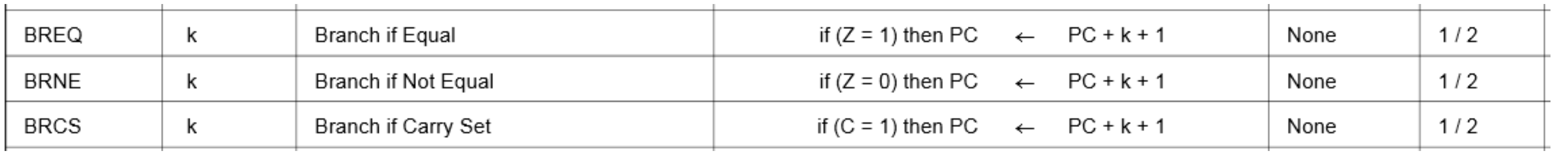


CLR	Rd	Clear Register	$Rd \leftarrow Rd \oplus Rd$	Z,N,V,S	1
SER	Rd	Set Register	$Rd \leftarrow \$FF$	None	1
MUL[1]	Rd,Rr	Multiply Unsigned	$R1{:}R0 \leftarrow Rd \times Rr$ (UU)	Z,C	2

- Given a clock frequency of 16 MHz and based on the above table a multiple MUL instruction will take

$2 \times 0.0625\ \mu s = 0.125\ \mu s$ to execute

- For branch instructions, the answer is not so straight forward.

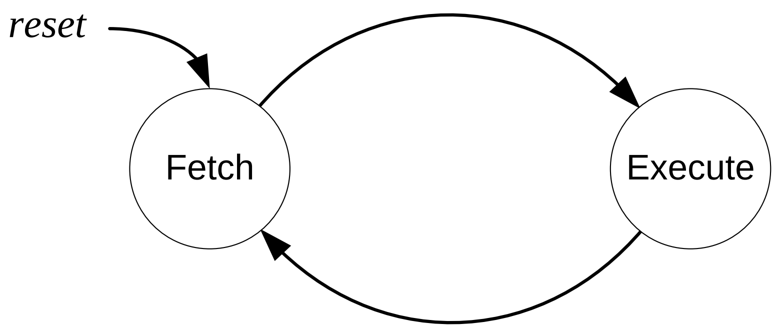


BREQ	k	Branch if Equal	if (Z = 1) then $PC \leftarrow PC + k + 1$	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then $PC \leftarrow PC + k + 1$	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then $PC \leftarrow PC + k + 1$	None	1 / 2

PIPELINING

Before you can fully understand branching and looping you need to understand the concept of pipelining and how it is implemented in our AVR processor.

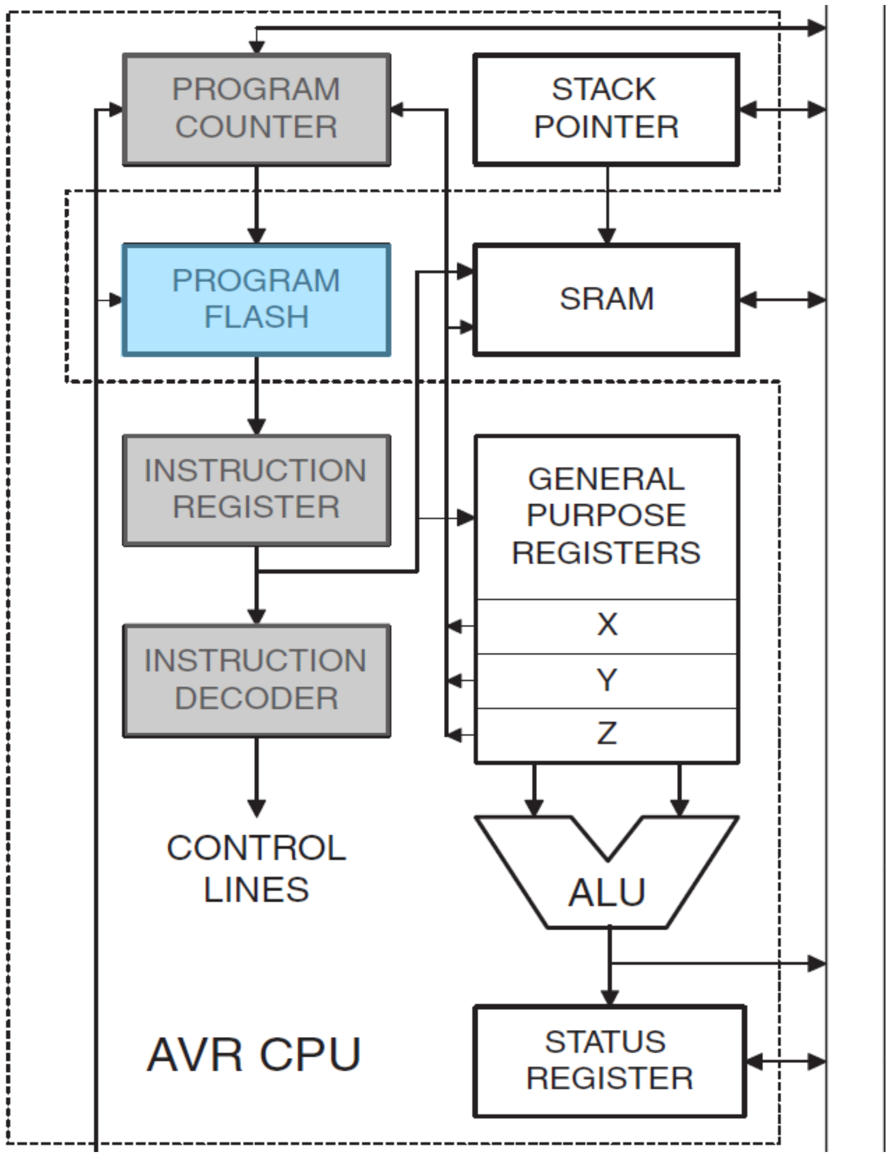
- Pipelining is a technique that breaks operations, such as instruction processing (**fetch and execute**) into smaller distinct stages so that a subsequent operation can begin before the previous one has completed.



- For most instructions, especially one based on a modified Harvard memory model, program memory is not accessed during the execution cycle. This memory down time could be used to fetch the next instruction to be executed, in parallel with the execution cycle of the current instruction. Here then is an opportunity for pipelining!

AVR Interstage Pipeline Registers

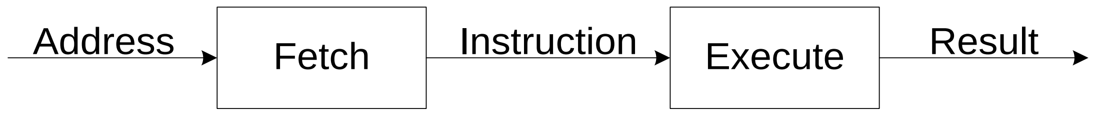
- A pipeline stage begins and ends with a register; controlled by a clock. Technically these are known as interstage pipeline registers.
- With respect to our AVR architecture the two registers of interest are the **Program Counter** (PC) and the **Instruction Register** (IR).
- Between the register(s) is combinational logic. Although counter-intuitive, **Flash Program** memory can be viewed as combinational logic with an address generating a word of data.
- Without pipelining these two registers in the control unit (PC, IR) would require two clock cycles to complete a basic computer operation cycle. Specifically, an instruction is (1) fetched and then (2) executed.



AVR Two-stage Instruction Pipeline

- The AVR pipeline has two independent stages. The first stage fetches an instruction and places it in the Instruction Register (IR), while the second stage is executing the instruction.

Fetch and Execute Cycle of the Atmel ATmega Microcontroller



- For our RISC architecture *most* instructions are executed in a single cycle (also known as elemental instructions). In this perfect world where all instructions take one cycle to fetch and one cycle to execute, after an initial delay of one cycle to fill the pipeline, known as *latency*, each instruction will take only one cycle to complete.

Program Execution in a AVR RISC two-Stage Instruction Pipelined Architecture

Time

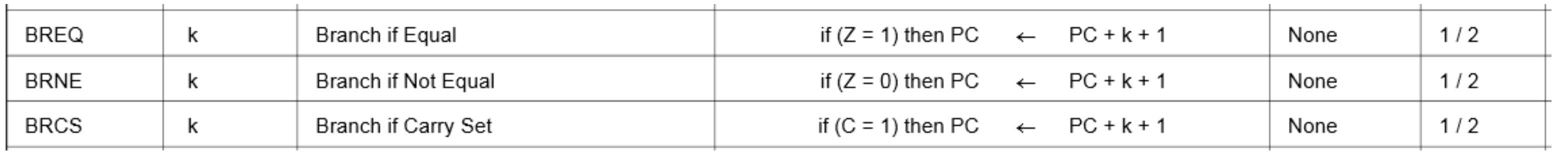
	1	2	3	4
Fetch	Instr. 1	Instr. 2	Instr. 3	Instr. 4
Execute		Instr. 1	Instr. 2	Instr. 3

Branch Penalty

- Within the context of pipeline architecture, when the execution stage of the pipeline is executing a conditional branch instruction, the execution stage must "predict" the outcome of the instruction in order to fetch what it "guesses" will be the next instruction.

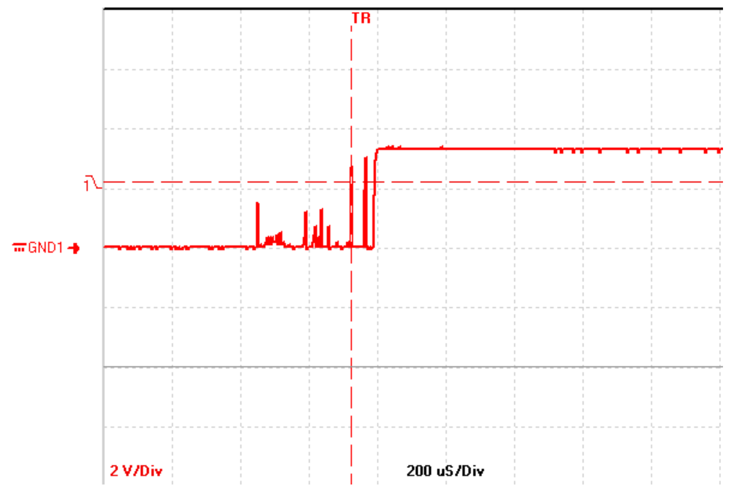
- While on average 80% of the time a branch is taken, the AVR always guesses that the branch will not be taken. This guess is made simply because it is the simplest to implement (the program counter automatically points at the next instruction to be executed).

- When a branch is taken, and the guess is wrong, the processor must build the pipeline from scratch thus accruing a "penalty." With our simple 2-stage pipeline that penalty is one clock cycle as shown in the AVR Instruction Set Document.



BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1 / 2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1 / 2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1 / 2

Button Debounce Example - Continued



[2]

- In the screen capture (red waveform), a button bounces for about 400us when pressed. Once the transition is detected, we want to design a software loop that will do nothing while the switch input stabilizes. To remove the noise, we will design a software delay routine that will generate a delay of approx. 500 us.

Delay Calculation for AVR

- Returning to our simple software loop

```
wait:
     ldi     r16, ____       // Loop Count
delay:
     dec     r16             //   1 clock cycle
     brne    delay           // + 2 cycles if true, 1 cycle if false
```

$T_{delay} = (N_{mc} \times L_{cnt} - 1)t_{mc}$

T_{delay} = Delay generated by the loop
t_{mc} = period of one machine cycle = $1/F_{clk}$ (note: 1 machine cycle = 1 clock cycle) = 1 / 16 MHz = 0.0625 usec
N_{mc} = number of machine cycles in 1 loop = 3 (*for brne Nmc = 2 cycles, we subtract 1 for the one cases where our guess is correct.*)
L_{cnt} = number of times loop is run (Loop Count) = ?

[2] http://generichid.sourceforge.net/buttonbounceDSO.png

Calculating Maximum Delay

Test is False

- Next we will calculate the maximum delay

  L_{cnt} = 0 which results in a count of 256

  T_{max_delay} = (3 x 256 - 1)(0.0625 µs) = 48 µsec (approx)

- Now Let's increase this delay by adding a `nop` instruction and then recalculating the maximum delay

  N_{mc} = number of machine cycles in 1 loop = 4

```
wait:
     clr     r16            //   0 = maximum delay
delay:
     nop                    //   1
     dec     r16            //   1 clock cycle
     brne    delay          // + 2 cycles if true, 1 cycle if false
```

  T_{max_delay} = (256 x 4 - 1)(0.0625 µs) = 64 µsec (approx) with r16 = 0 (`clr r16`)

Calculating Loop Count for a Given Delay

- To generate a delay of 500 µs we will initialize r16 for a delay of 50 µs and then write an outside loop that will run the inside loop 10 times for a total delay of approximately 500 µs

- Solving our T_{max} equation for Loop Count L_{cnt}

  $L_{cnt} = (T_{delay}/t_{mc} + 1)/N_{mc} = (T_{delay} \times F_{clk} + 1)/N_{mc}$

- Set L_{cnt} for a delay of 50 µsec

  $L_{cnt} = (50\mu s/0.0625\mu s + 1)/4 \cong 200 = 0xC8$

```
wait:
     ldi     r16, 0xC8      // 200
delay:
     nop                    //   1
     dec     r16            //   1 clock cycle
     brne    delay          // + 2 cycles if true, 1 cycle if false
```

Loop inside a loop delay

- On your own, create an outside loop with a count of 10 to give us a delay of approximately 500 µsec (Hint see Example 3-18 in your textbook)

Design Example with EE346 Shield

When the user presses the button, read first 3 switches (least significant), if the number is less than or equal to 5 then calculate factorial. If greater than 5 turn on decimal point. Display the least significant 4 bits of the answer.

My Design Steps

Step 1: Initialized Ports

Step 2: Turned on LED 0 to indicate initialization complete

Step 3: Wrote code to pulse the clock

Step 4: Read in pin waiting for button to be pressed (**Loop Example 1**)

Step 5: Need to filter out Bounce (**Loop Example 2**)

Maximum delay that could be generated was only 48 usec

Step 6: Added a NOP instruction, max delay was now 64 usec

Set delay for nice even number of 50 usec

Step 7: Made an outside loop of 10 (**Loop Example 3**)

Step 8: Converted loop to a subroutine so I could change condition to button release.

Step 9: Check for button pressed and then released

Step 10: Read Switch and check if less than or equal to 6

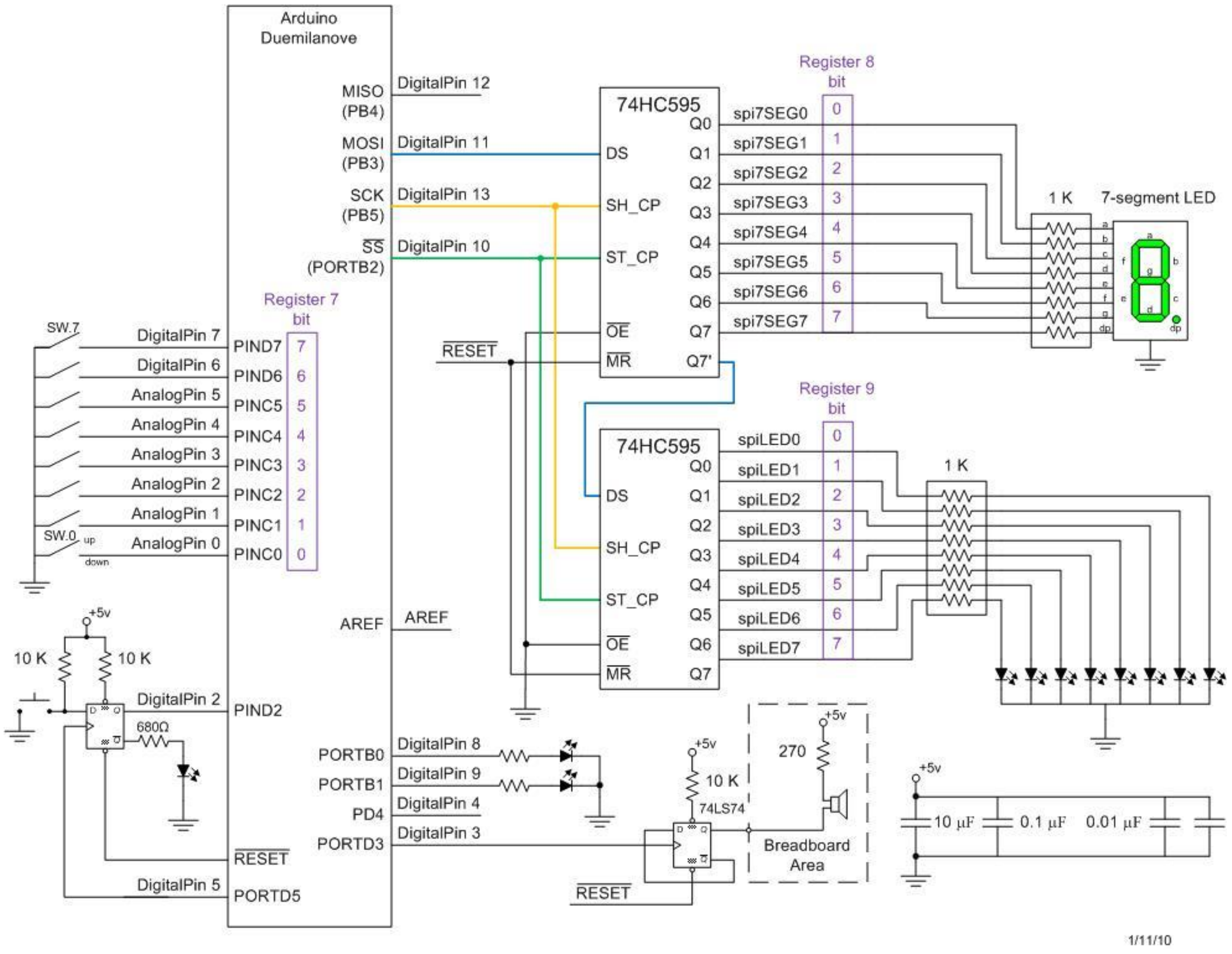
Step 11: Calculate Factorial (**Loop Example 4**)

Step 12: Store 4 digit answer to SRAM (SRAM **Indirect Addressing Mode**)

Step 13: Sequentially, Load each digit and ... (SRAM **Indirect Addressing Mode**)

Step 14: convert to 7-segment display (Flash Program **Indirect Addressing Mode**)

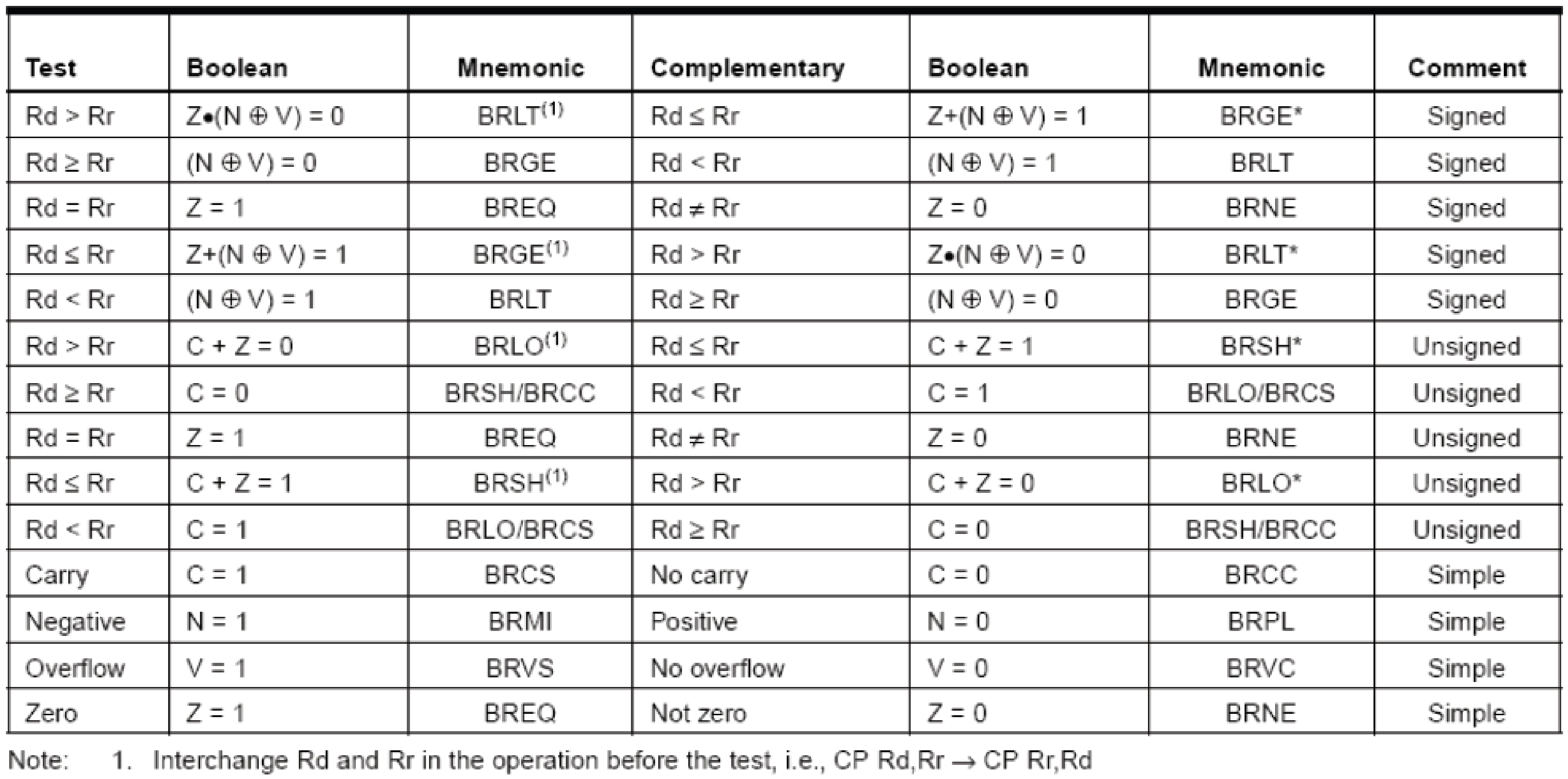
CSULB Proto-Shield Schematic



Configure GPIO Ports

DDxn	PORTxn	PUD (in MCUCR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Conditional Branch Summary

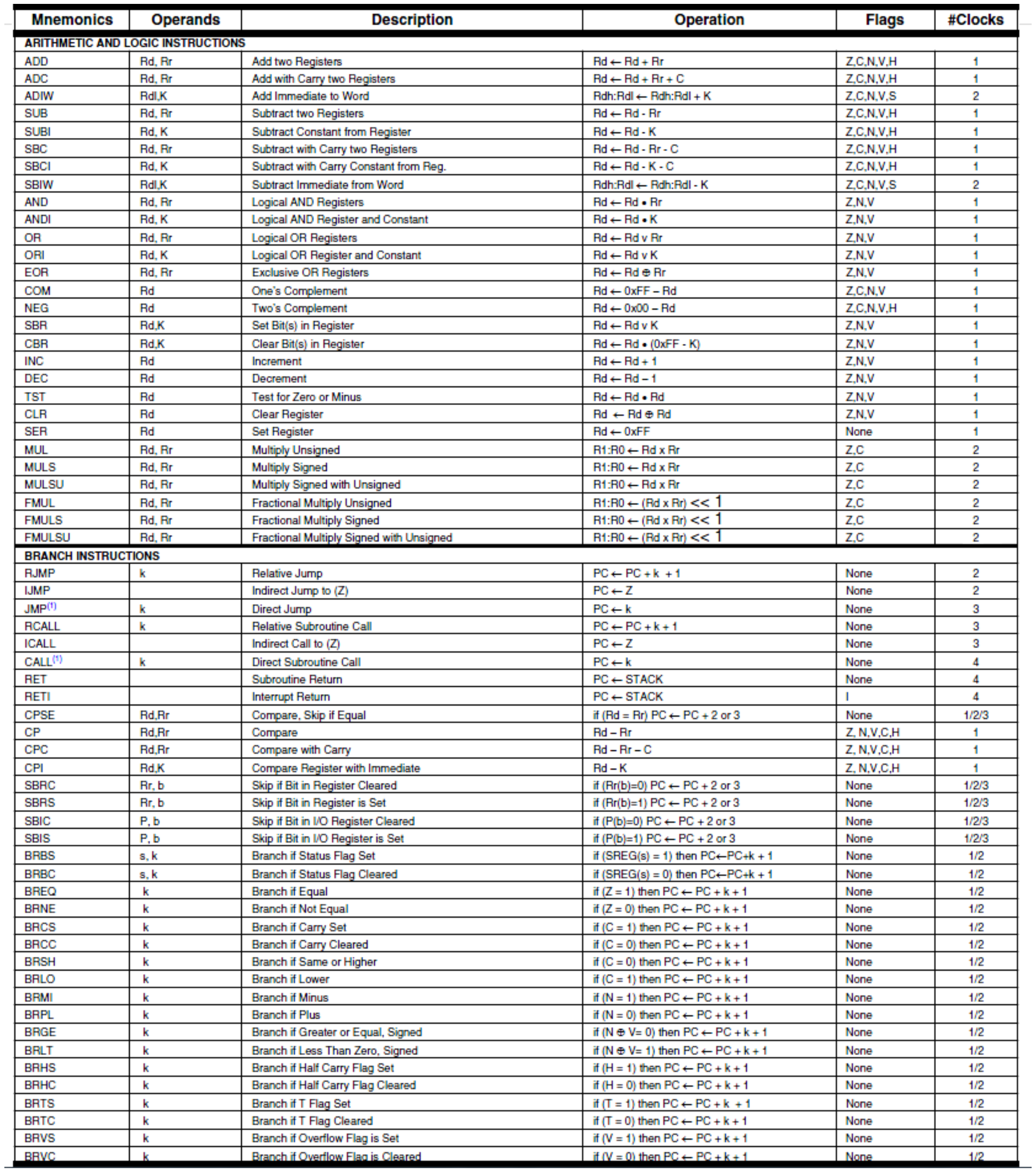


Test	Boolean	Mnemonic	Complementary	Boolean	Mnemonic	Comment
$Rd > Rr$	$Z \cdot (N \oplus V) = 0$	BRLT[1]	$Rd \leq Rr$	$Z + (N \oplus V) = 1$	BRGE*	Signed
$Rd \geq Rr$	$(N \oplus V) = 0$	BRGE	$Rd < Rr$	$(N \oplus V) = 1$	BRLT	Signed
$Rd = Rr$	$Z = 1$	BREQ	$Rd \neq Rr$	$Z = 0$	BRNE	Signed
$Rd \leq Rr$	$Z + (N \oplus V) = 1$	BRGE[1]	$Rd > Rr$	$Z \cdot (N \oplus V) = 0$	BRLT*	Signed
$Rd < Rr$	$(N \oplus V) = 1$	BRLT	$Rd \geq Rr$	$(N \oplus V) = 0$	BRGE	Signed
$Rd > Rr$	$C + Z = 0$	BRLO[1]	$Rd \leq Rr$	$C + Z = 1$	BRSH*	Unsigned
$Rd \geq Rr$	$C = 0$	BRSH/BRCC	$Rd < Rr$	$C = 1$	BRLO/BRCS	Unsigned
$Rd = Rr$	$Z = 1$	BREQ	$Rd \neq Rr$	$Z = 0$	BRNE	Unsigned
$Rd \leq Rr$	$C + Z = 1$	BRSH[1]	$Rd > Rr$	$C + Z = 0$	BRLO*	Unsigned
$Rd < Rr$	$C = 1$	BRLO/BRCS	$Rd \geq Rr$	$C = 0$	BRSH/BRCC	Unsigned
Carry	$C = 1$	BRCS	No carry	$C = 0$	BRCC	Simple
Negative	$N = 1$	BRMI	Positive	$N = 0$	BRPL	Simple
Overflow	$V = 1$	BRVS	No overflow	$V = 0$	BRVC	Simple
Zero	$Z = 1$	BREQ	Not zero	$Z = 0$	BRNE	Simple

Note: 1. Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr $\rightarrow$ CP Rr,Rd

Source: http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf page 10
http://apachepersonal.miun.se/~mathje/ET014G/Lectures/F3-AVR.pdf

ATmega328P Instruction Set[3]



Mnemonics	Operands	Description	Operation	Flags	#Clocks
ARITHMETIC AND LOGIC INSTRUCTIONS					
ADD	Rd, Rr	Add two Registers	Rd ← Rd + Rr	Z,C,N,V,H	1
ADC	Rd, Rr	Add with Carry two Registers	Rd ← Rd + Rr + C	Z,C,N,V,H	1
ADIW	Rdl,K	Add Immediate to Word	Rdh:Rdl ← Rdh:Rdl + K	Z,C,N,V,S	2
SUB	Rd, Rr	Subtract two Registers	Rd ← Rd - Rr	Z,C,N,V,H	1
SUBI	Rd, K	Subtract Constant from Register	Rd ← Rd - K	Z,C,N,V,H	1
SBC	Rd, Rr	Subtract with Carry two Registers	Rd ← Rd - Rr - C	Z,C,N,V,H	1
SBCI	Rd, K	Subtract with Carry Constant from Reg.	Rd ← Rd - K - C	Z,C,N,V,H	1
SBIW	Rdl,K	Subtract Immediate from Word	Rdh:Rdl ← Rdh:Rdl - K	Z,C,N,V,S	2
AND	Rd, Rr	Logical AND Registers	Rd ← Rd • Rr	Z,N,V	1
ANDI	Rd, K	Logical AND Register and Constant	Rd ← Rd • K	Z,N,V	1
OR	Rd, Rr	Logical OR Registers	Rd ← Rd v Rr	Z,N,V	1
ORI	Rd, K	Logical OR Register and Constant	Rd ← Rd v K	Z,N,V	1
EOR	Rd, Rr	Exclusive OR Registers	Rd ← Rd ⊕ Rr	Z,N,V	1
COM	Rd	One's Complement	Rd ← 0xFF – Rd	Z,C,N,V	1
NEG	Rd	Two's Complement	Rd ← 0x00 – Rd	Z,C,N,V,H	1
SBR	Rd,K	Set Bit(s) in Register	Rd ← Rd v K	Z,N,V	1
CBR	Rd,K	Clear Bit(s) in Register	Rd ← Rd • (0xFF - K)	Z,N,V	1
INC	Rd	Increment	Rd ← Rd + 1	Z,N,V	1
DEC	Rd	Decrement	Rd ← Rd – 1	Z,N,V	1
TST	Rd	Test for Zero or Minus	Rd ← Rd • Rd	Z,N,V	1
CLR	Rd	Clear Register	Rd ← Rd ⊕ Rd	Z,N,V	1
SER	Rd	Set Register	Rd ← 0xFF	None	1
MUL	Rd, Rr	Multiply Unsigned	R1:R0 ← Rd x Rr	Z,C	2
MULS	Rd, Rr	Multiply Signed	R1:R0 ← Rd x Rr	Z,C	2
MULSU	Rd, Rr	Multiply Signed with Unsigned	R1:R0 ← Rd x Rr	Z,C	2
FMUL	Rd, Rr	Fractional Multiply Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULS	Rd, Rr	Fractional Multiply Signed	R1:R0 ← (Rd x Rr) << 1	Z,C	2
FMULSU	Rd, Rr	Fractional Multiply Signed with Unsigned	R1:R0 ← (Rd x Rr) << 1	Z,C	2
BRANCH INSTRUCTIONS					
RJMP	k	Relative Jump	PC ← PC + k + 1	None	2
IJMP		Indirect Jump to (Z)	PC ← Z	None	2
JMP(1)	k	Direct Jump	PC ← k	None	3
RCALL	k	Relative Subroutine Call	PC ← PC + k + 1	None	3
ICALL		Indirect Call to (Z)	PC ← Z	None	3
CALL(1)	k	Direct Subroutine Call	PC ← k	None	4
RET		Subroutine Return	PC ← STACK	None	4
RETI		Interrupt Return	PC ← STACK	I	4
CPSE	Rd,Rr	Compare, Skip if Equal	if (Rd = Rr) PC ← PC + 2 or 3	None	1/2/3
CP	Rd,Rr	Compare	Rd – Rr	Z, N,V,C,H	1
CPC	Rd,Rr	Compare with Carry	Rd – Rr – C	Z, N,V,C,H	1
CPI	Rd,K	Compare Register with Immediate	Rd – K	Z, N,V,C,H	1
SBRC	Rr, b	Skip if Bit in Register Cleared	if (Rr(b)=0) PC ← PC + 2 or 3	None	1/2/3
SBRS	Rr, b	Skip if Bit in Register is Set	if (Rr(b)=1) PC ← PC + 2 or 3	None	1/2/3
SBIC	P, b	Skip if Bit in I/O Register Cleared	if (P(b)=0) PC ← PC + 2 or 3	None	1/2/3
SBIS	P, b	Skip if Bit in I/O Register is Set	if (P(b)=1) PC ← PC + 2 or 3	None	1/2/3
BRBS	s, k	Branch if Status Flag Set	if (SREG(s) = 1) then PC←PC+k + 1	None	1/2
BRBC	s, k	Branch if Status Flag Cleared	if (SREG(s) = 0) then PC←PC+k + 1	None	1/2
BREQ	k	Branch if Equal	if (Z = 1) then PC ← PC + k + 1	None	1/2
BRNE	k	Branch if Not Equal	if (Z = 0) then PC ← PC + k + 1	None	1/2
BRCS	k	Branch if Carry Set	if (C = 1) then PC ← PC + k + 1	None	1/2
BRCC	k	Branch if Carry Cleared	if (C = 0) then PC ← PC + k + 1	None	1/2
BRSH	k	Branch if Same or Higher	if (C = 0) then PC ← PC + k + 1	None	1/2
BRLO	k	Branch if Lower	if (C = 1) then PC ← PC + k + 1	None	1/2
BRMI	k	Branch if Minus	if (N = 1) then PC ← PC + k + 1	None	1/2
BRPL	k	Branch if Plus	if (N = 0) then PC ← PC + k + 1	None	1/2
BRGE	k	Branch if Greater or Equal, Signed	if (N ⊕ V= 0) then PC ← PC + k + 1	None	1/2
BRLT	k	Branch if Less Than Zero, Signed	if (N ⊕ V= 1) then PC ← PC + k + 1	None	1/2
BRHS	k	Branch if Half Carry Flag Set	if (H = 1) then PC ← PC + k + 1	None	1/2
BRHC	k	Branch if Half Carry Flag Cleared	if (H = 0) then PC ← PC + k + 1	None	1/2
BRTS	k	Branch if T Flag Set	if (T = 1) then PC ← PC + k + 1	None	1/2
BRTC	k	Branch if T Flag Cleared	if (T = 0) then PC ← PC + k + 1	None	1/2
BRVS	k	Branch if Overflow Flag is Set	if (V = 1) then PC ← PC + k + 1	None	1/2
BRVC	k	Branch if Overflow Flag is Cleared	if (V = 0) then PC ← PC + k + 1	None	1/2

[3] Source: ATmega328P Data Sheet http://www.atmel.com/dyn/resources/prod_documents/8161S.pdf Chapter 31 Instruction Set Summary

Mnemonics	Operands	Description	Operation	Flags	#Clocks
BRIE	k	Branch if Interrupt Enabled	if (I = 1) then PC ← PC + k + 1	None	1/2
BRID	k	Branch if Interrupt Disabled	if (I = 0) then PC ← PC + k + 1	None	1/2
BIT AND BIT-TEST INSTRUCTIONS					
SBI	P,b	Set Bit in I/O Register	I/O(P,b) ← 1	None	2
CBI	P,b	Clear Bit in I/O Register	I/O(P,b) ← 0	None	2
LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n), Rd(0) ← 0	Z,C,N,V	1
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1), Rd(7) ← 0	Z,C,N,V	1
ROL	Rd	Rotate Left Through Carry	Rd(0)←C,Rd(n+1)← Rd(n),C←Rd(7)	Z,C,N,V	1
ROR	Rd	Rotate Right Through Carry	Rd(7)←C,Rd(n)← Rd(n+1),C←Rd(0)	Z,C,N,V	1
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6	Z,C,N,V	1
SWAP	Rd	Swap Nibbles	Rd(3..0)←Rd(7..4),Rd(7..4)←Rd(3..0)	None	1
BSET	s	Flag Set	SREG(s) ← 1	SREG(s)	1
BCLR	s	Flag Clear	SREG(s) ← 0	SREG(s)	1
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)	T	1
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T	None	1
SEC		Set Carry	C ← 1	C	1
CLC		Clear Carry	C ← 0	C	1
SEN		Set Negative Flag	N ← 1	N	1
CLN		Clear Negative Flag	N ← 0	N	1
SEZ		Set Zero Flag	Z ← 1	Z	1
CLZ		Clear Zero Flag	Z ← 0	Z	1
SEI		Global Interrupt Enable	I ← 1	I	1
CLI		Global Interrupt Disable	I ← 0	I	1
SES		Set Signed Test Flag	S ← 1	S	1
CLS		Clear Signed Test Flag	S ← 0	S	1
SEV		Set Twos Complement Overflow.	V ← 1	V	1
CLV		Clear Twos Complement Overflow	V ← 0	V	1
SET		Set T in SREG	T ← 1	T	1
CLT		Clear T in SREG	T ← 0	T	1
SEH		Set Half Carry Flag in SREG	H ← 1	H	1
CLH		Clear Half Carry Flag in SREG	H ← 0	H	1
DATA TRANSFER INSTRUCTIONS					
MOV	Rd, Rr	Move Between Registers	Rd ← Rr	None	1
MOVW	Rd, Rr	Copy Register Word	Rd+1:Rd ← Rr+1:Rr	None	1
LDI	Rd, K	Load Immediate	Rd ← K	None	1
LD	Rd, X	Load Indirect	Rd ← (X)	None	2
LD	Rd, X+	Load Indirect and Post-Inc.	Rd ← (X), X ← X + 1	None	2
LD	Rd, - X	Load Indirect and Pre-Dec.	X ← X - 1, Rd ← (X)	None	2
LD	Rd, Y	Load Indirect	Rd ← (Y)	None	2
LD	Rd, Y+	Load Indirect and Post-Inc.	Rd ← (Y), Y ← Y + 1	None	2
LD	Rd, - Y	Load Indirect and Pre-Dec.	Y ← Y - 1, Rd ← (Y)	None	2
LDD	Rd,Y+q	Load Indirect with Displacement	Rd ← (Y + q)	None	2
LD	Rd, Z	Load Indirect	Rd ← (Z)	None	2
LD	Rd, Z+	Load Indirect and Post-Inc.	Rd ← (Z), Z ← Z+1	None	2
LD	Rd, -Z	Load Indirect and Pre-Dec.	Z ← Z - 1, Rd ← (Z)	None	2
LDD	Rd, Z+q	Load Indirect with Displacement	Rd ← (Z + q)	None	2
LDS	Rd, k	Load Direct from SRAM	Rd ← (k)	None	2
ST	X, Rr	Store Indirect	(X) ← Rr	None	2
ST	X+, Rr	Store Indirect and Post-Inc.	(X) ← Rr, X ← X + 1	None	2
ST	- X, Rr	Store Indirect and Pre-Dec.	X ← X - 1, (X) ← Rr	None	2
ST	Y, Rr	Store Indirect	(Y) ← Rr	None	2
ST	Y+, Rr	Store Indirect and Post-Inc.	(Y) ← Rr, Y ← Y + 1	None	2
ST	- Y, Rr	Store Indirect and Pre-Dec.	Y ← Y - 1, (Y) ← Rr	None	2
STD	Y+q,Rr	Store Indirect with Displacement	(Y + q) ← Rr	None	2
ST	Z, Rr	Store Indirect	(Z) ← Rr	None	2
ST	Z+, Rr	Store Indirect and Post-Inc.	(Z) ← Rr, Z ← Z + 1	None	2
ST	-Z, Rr	Store Indirect and Pre-Dec.	Z ← Z - 1, (Z) ← Rr	None	2
STD	Z+q,Rr	Store Indirect with Displacement	(Z + q) ← Rr	None	2
STS	k, Rr	Store Direct to SRAM	(k) ← Rr	None	2
LPM		Load Program Memory	R0 ← (Z)	None	3
LPM	Rd, Z	Load Program Memory	Rd ← (Z)	None	3
LPM	Rd, Z+	Load Program Memory and Post-Inc	Rd ← (Z), Z ← Z+1	None	3
SPM		Store Program Memory	(Z) ← R1:R0	None	-
IN	Rd, P	In Port	Rd ← P	None	1
OUT	P, Rr	Out Port	P ← Rr	None	1
PUSH	Rr	Push Register on Stack	STACK ← Rr	None	2

Mnemonics	Operands	Description	Operation	Flags	#Clocks
POP	Rd	Pop Register from Stack	Rd ← STACK	None	2
MCU CONTROL INSTRUCTIONS					
NOP		No Operation		None	1
SLEEP		Sleep	(see specific descr. for Sleep function)	None	1
WDR		Watchdog Reset	(see specific descr. for WDR/timer)	None	1
BREAK		Break	For On-chip Debug Only	None	N/A

Note: 1. These instructions are only available in ATmega168PA and ATmega328P.