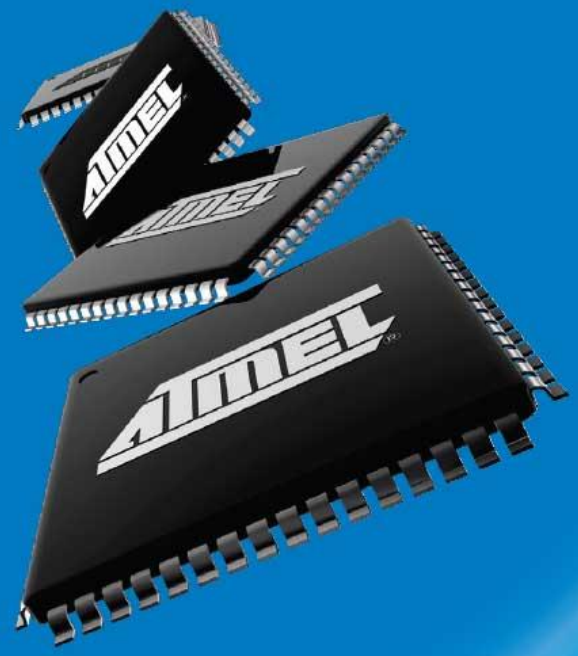
AVR®

8-bit Microcontrollers

AVR®32

32-bit Microcontrollers and Application Processors



Introduction to AVR Assembly Language Programming II

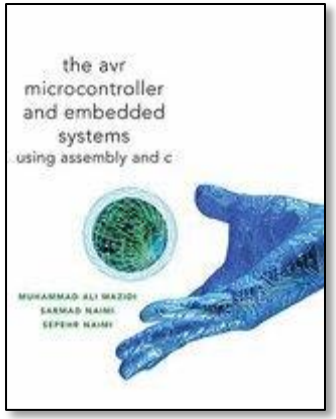
February 2009

Everywhere You Are®

AVR Control Transfer -AVR Branching

Reading

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



Chapter 3: Branch, Call, and Time Delay Loop

Section 3.1: Branching and Looping (*Branch Only*)

Additional Reading

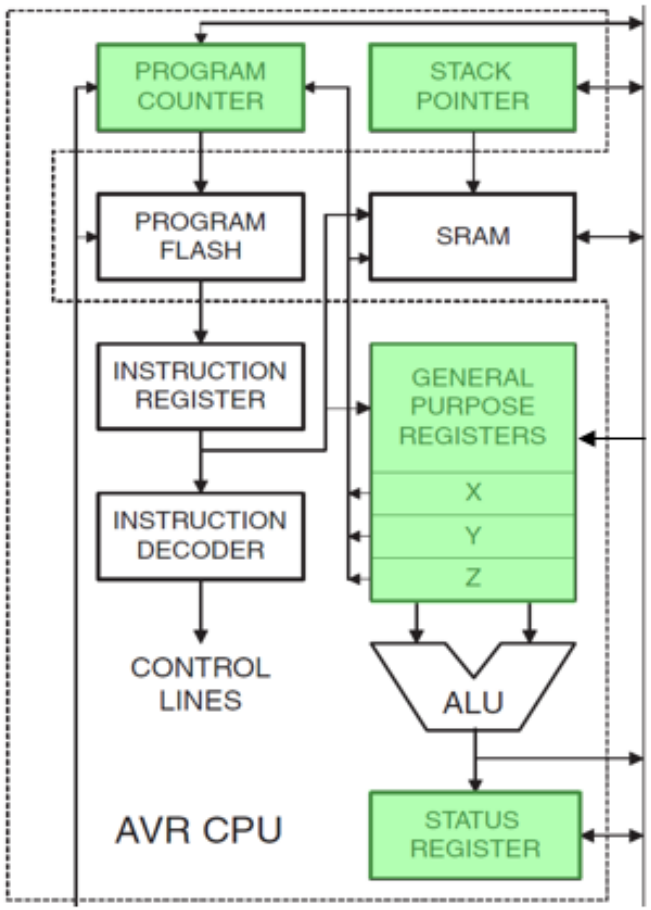
- Introduction to AVR assembler programming for beginners, controlling sequential execution of the program http://www.avr-asm-tutorial.net/avr_en/beginner/JUMP.html
- AVR Assembler User Guide http://www.atmel.com/dyn/resources/prod documents/doc1022.pdf

TABLE OF CONTENTS

Instruction Set Architecture (Review) 4
Instruction Set (*Review*) 5
Jump Instructions 6
How the Direct Unconditional Control Transfer Instructions jmp and call Work 7
How the Relative Unconditional Control Transfer Instructions rjmp and rcall Work 8
Branch Instructions 9
How the Relative Conditional Control Transfer Instruction BREQ Works 10
Conditional Branch Encoding 12
A Conditional Control Transfer (Branch) Sequence 13
Conditional Branch Instruction Summary 14
Implementing a High-Level IF Statement 16
Implementing a High-Level IF…ELSE Statement 17
Assembly Optimization of a High-Level IF…ELSE Statement – Advanced Topic – 18
Program Examples 19
Appendix A: Control Transfer Instruction Encoding 27
Appendix B – AVR Status Register (SREG) 30
Appendix C – Control Transfer (Branch) Instructions 31
Appendix D – ATmega328P Instruction Set 32

INSTRUCTION SET ARCHITECTURE (REVIEW)

The *Instruction Set Architecture* (ISA) of a microprocessor includes all the registers that are accessible to the programmer. In other words, registers that can be modified by the *instruction set* of the processor. With respect to the AVR CPU illustrated here[1], these ISA registers include the 32 x 8-bit general purpose resisters, status resister (SREG), the stack pointer (SP), and the program counter (PC).



Data Transfer instructions are used to load and store data to the General Purpose Registers, also known as the *Register File*. Exceptions are the push and pop instructions which modify the Stack Pointer. By definition these instructions do not modify the status register (SREG).

Arithmetic and Logic Instructions plus Bit and Bit-Test Instructions use the ALU to operate on the data contained in the general purpose registers. Flags contained in the status register (SREG) provide important information concerning the results of these operations. For example, if you are adding two signed numbers together, you will want to know if the answer is correct. The state of the overflow flag (OV) bit within SREG gives you the answer to this question (1 = error, 0 no error).

Control Transfer Instructions allow you to change the contents of the PC either conditionally or unconditionally. Continuing our example if an error results from adding two signed numbers together we may want to conditionally (OV = 1) branch to an error handling routine. As the AVR processor fetches and executes instructions it automatically increments the program counter (PC) so it always points at the **next instruction to be executed**.

[1] Source: ATmega16 Data Sheet http://www.atmel.com/dyn/resources/prod_documents/2466s.pdf page 3 Figure 1-5 "AVR Central Processing Unit ISA Registers"

INSTRUCTION SET (*REVIEW*)

The ***Instruction Set*** of our AVR processor can be functionally divided (or classified) into the following parts:

- Data Transfer Instructions
- Arithmetic and Logic Instructions
- Bit and Bit-Test Instructions
- **Control Transfer (Branch) Instructions**
- MCU Control Instructions

JUMP INSTRUCTIONS

- There are two basic types of control transfer instructions – Unconditional and Conditional.
- From a programmer's perspective an unconditional or jump instruction, jumps to the label specified. For example, `jmp loop` will unconditionally jump to the label loop in your program.
- Here are the unconditional control transfer "Jump" instructions of the AVR processor

  - Direct `jmp, call`
  - Relative (1) `rjmp, rcall`
  - Indirect `ijmp, icall`
  - Subroutine & Interrupt Return `ret, reti`

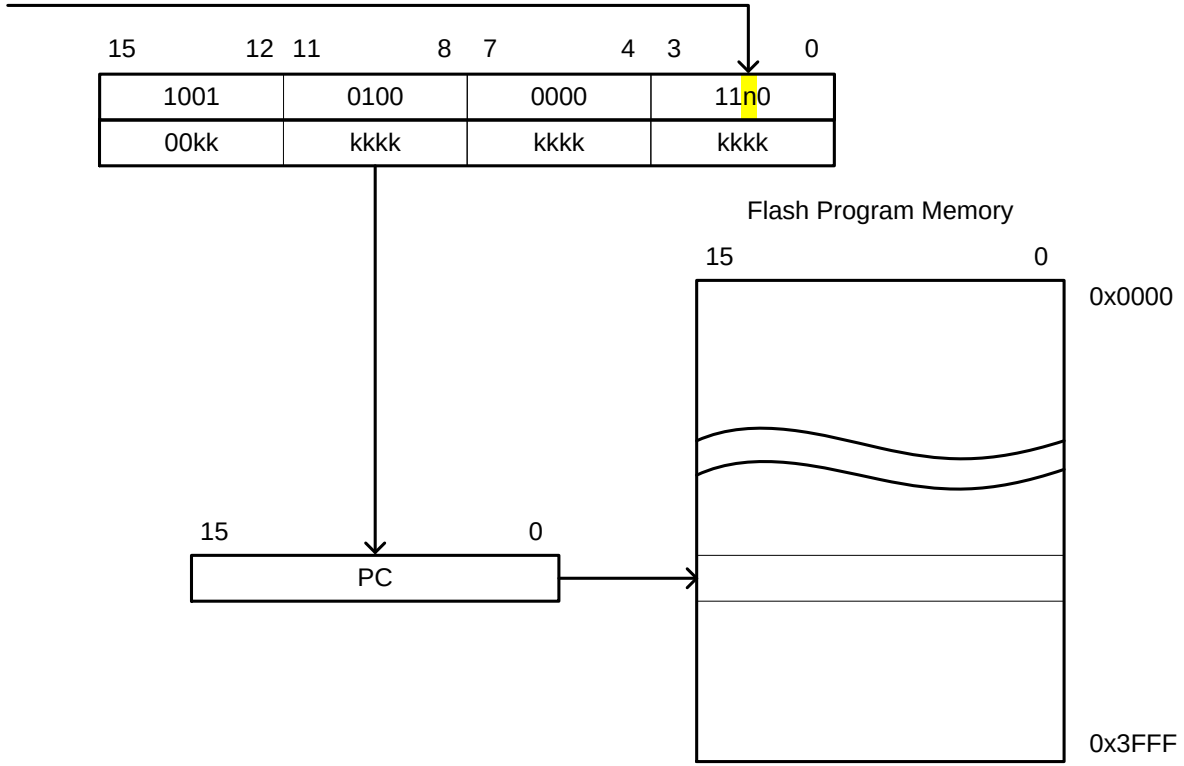
Note:

1. Jump relative to PC + (-2^{k-1} ⇨ $2^{k-1}-1$, where k = 12) ⇨PC-2048 to PC+2047, within 16 K word address space of ATmega328P

How the Direct Unconditional Control Transfer Instructions jmp and call Work

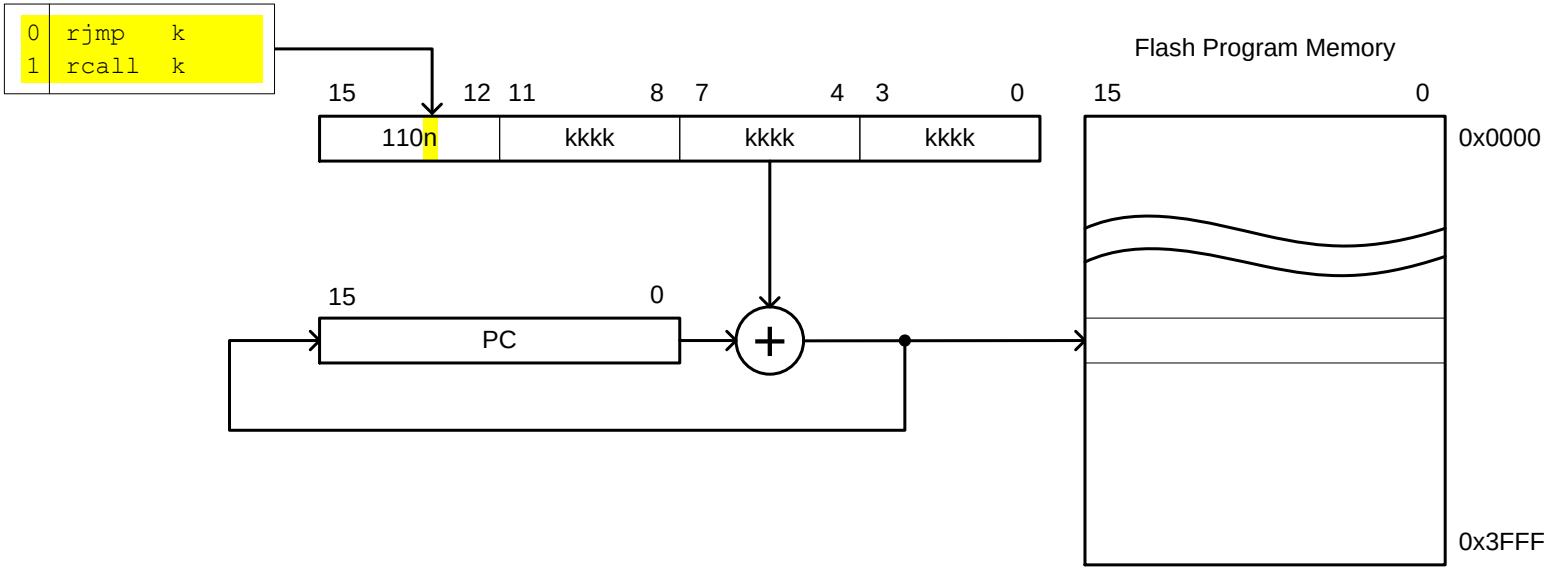
- From a computer engineer's perspective, a direct jump is accomplished by loading the target address into the program counter (PC). In the example, the target address is equated to label "`loop`."
    - To provide a more concrete example, assume the label `loop` corresponds to address `0x0123` in Flash Program Memory.
    - To execute this instruction, the control logic of central procession unit (CPU) loads the 16-bit Program Counter (PC) register with `0x123`.
    - Consequently, on the next fetch cycle it is the instruction at location 0x0123 that is fetched and then executed. ***Control*** *of the program has been* ***transferred*** *to this address.*

```
0  jmp   k
1  call  k
```



How the Relative Unconditional Control Transfer Instructions rjmp and rcall Work

- From a computer engineer's perspective, a relative jump is accomplished by adding a 12-bit signed offset to the program counter (PC)[2]. The result corresponding to the target address. In the example, the target address is equated to label "`loop`."
    - To provide a more concrete example, assume the label `loop` corresponds to address `0x0123` in Flash Program Memory (the target address).
    - An `rjmp loop` instruction is located at address 0x206. When the `rjmp` is executed, the PC is currently fetching what it thinks is the next instruction to be executed at address 0x207.
    - To accomplish this jump the relative address (kkkk kkkk kkkk) is equal to 0xF1C (i.e., 0x123 – 0x207).
    - Consequently, on the next fetch cycle it is the instruction at location 0x0123 that is fetched and then executed. ***Control*** *of the program has been* ***Transferred*** *to this address*[3].



[2] In the language of Computer Engineering, we are exploiting spatial locality of reference.

[3] The instruction at address 0x207 is not executed

BRANCH INSTRUCTIONS

- When a conditional or branch instruction is executed one of two things may happen.

  1. If the test condition is true then the branch will be taken (see jump instructions).

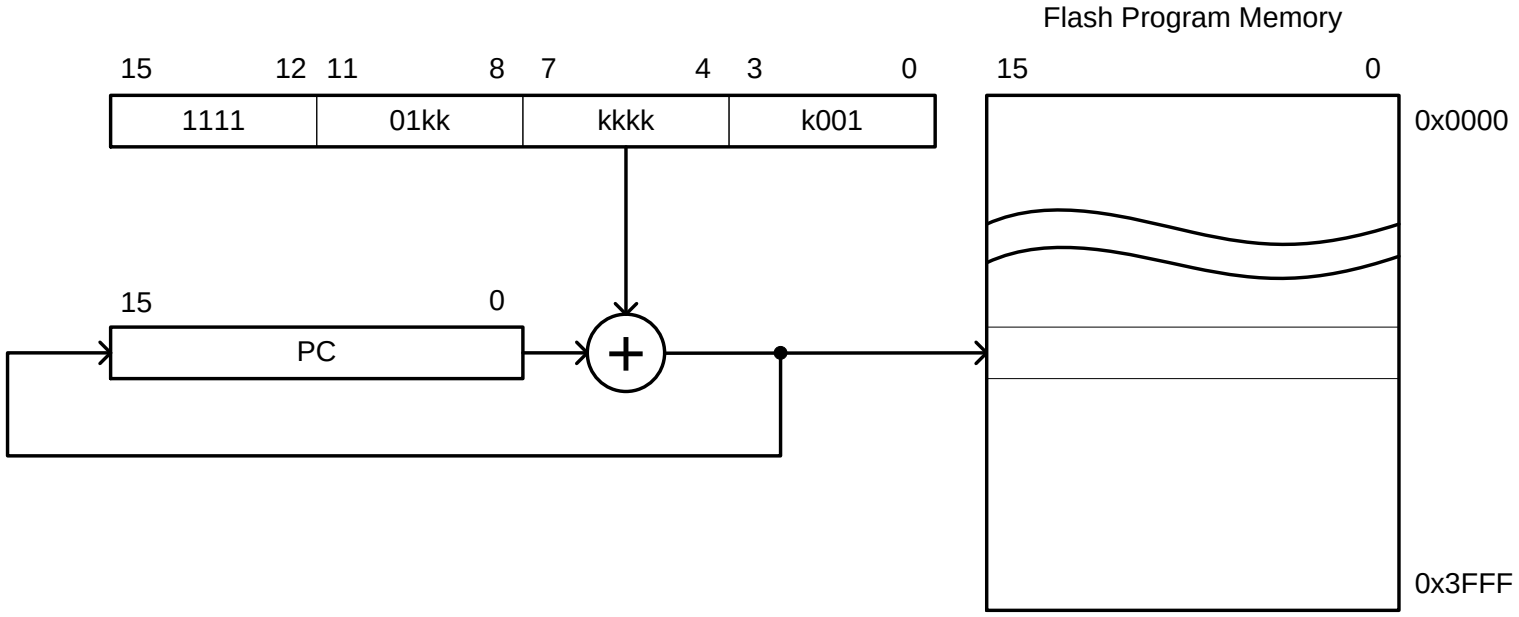
  2. If the test condition is false then nothing happens (see `nop` instruction).

     - This statement is not entirely accurate. Because the program counter always points to the next instruction to be executed, during the execution state, doing nothing means fetching the next instruction.

- The "test condition" is a function of one or more SREG flag bits. For example, while the Branch if equal (`breq`) or not equal (`brne`) instructions test only the Z flag; instructions like branch if less than (`brlt`) and branch if greater than or equal (`brge`) test the condition of the Z, N, and V flag bits.

How the Relative Conditional Control Transfer Instruction BREQ Works

- If a relative branch is taken (test condition is true) a 7-bit signed offset is added to the PC. The result corresponding to the target address. In the example, the target address is equated to label "`match`."
  - To provide a more concrete example, assume the label `nomatch` corresponds to address `0x0123` in Flash Program Memory (the target address).
  - A `brne nomatch` instruction is located at address `0x0112`. When the `brne` instruction is executed, the PC is currently fetching what it thinks is the next instruction to be executed at address `0x0113`.
  - To accomplish this jump the relative address (kk kkkk) is equal to `0b01_0000` (i.e., `0x123 – 0x113`).
  - Consequently, on the next fetch cycle it is the instruction at location `0x0123` that is fetched and then executed. ***Control*** *of the program has been* ***Transferred*** *to this address*[4].



[4] Because in our example, the test condition is false (Z = 0) the instruction at address 0x113 is not executed.

BRANCH INSTRUCTIONS

I	T	H	S	V	N	Z	C
-	-	-	-	-	-	-	-

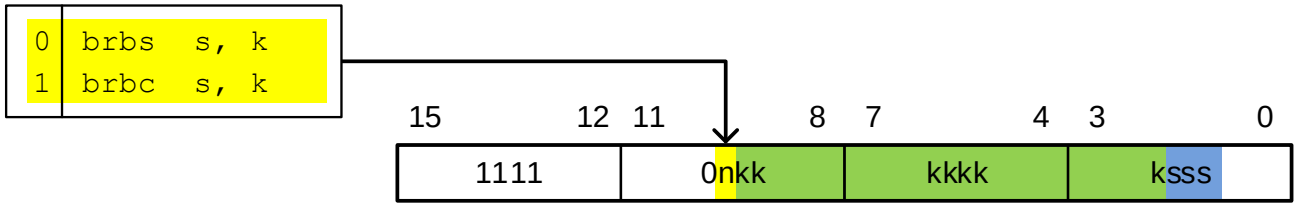
- All conditional branch instructions may be implemented as `brbs s,k` or `brbc s,k`, where s is the bit number of the SREG flag bit. For example `brbs 6, bitset` would branch to label `bitset`, if the SREG T bit was set.
- To make your code more readable, the AVR assembler adds the following "alias" instructions.
  - SREG Flag bit is clear (`brFlagc`) or set (`brFlags`) by name (I, **T**, **H**, S, **V**, N, Z, **C**) or bit (`brbc`, `brbs`).
  - These SREG flag bits (I, T, H, **S**, V, **N**, **Z**, **C**) use more descriptive mnemonics.
    - ✓ Branch if equal (`breq`) or not equal (`brne`) test the Z flag.
    - ✓ Unsigned arithmetic branch if plus (`brpl`) or minus (`brmi`) test the N flag, while branch if same or higher (`brsh`) or lower (`brlo`), test the C flag and are equivalent to `brcc` and `brcs` respectively.
    - ✓ Signed 2's complement arithmetic branch if number is less than zero (`brlt`) or greater than or equal to zero (`brge`) test the S flag
- Skip if …
  - Bit (b) in a register is clear (`sbrc`) or set (`sbrs`).
  - Bit (b) in I/O register is clear (`sbic`) or set (`sbis`). Limited to I/O addresses 0-31

Note:

1. All branch instructions are relative to PC + (-2^{k-1} ⇨ $2^{k-1}-1$, where k = 7) + 1 ⇨PC-64 to PC+63
2. Skip instructions may take 1, 2, or 3 cycles depending if the skip is not taken, and the number of Flash program memory words in the instruction to be skipped (1 or 2).

CONDITIONAL BRANCH ENCODING

- Here is how the `brbs, brbc` and their alias assembly instructions are encoded.



		alias	
SREG	sss	brbs s, k	brbc s, k
I	111	brie k	brid k
T	110	brts k	brtc k
H	101	brhs k	brhc k
S	100	brlt k	brge k
V	011	brvs k	brvc k
N	010	brmi k	brpl k
Z	001	breq k	brne k
C	000	brcs k brlo k	brcc k brsh k

A Conditional Control Transfer (Branch) Sequence

- A conditional control transfer (branch) sequence is typically comprised of 2 instructions.
  1. The first instruction performs some arithmetic or logic operation using the ALU of the processor.
     - Examples of this first type of instruction includes: `cp, cpc, cpi, tst`
     - These ALU operations result in SREG flag bits 5 to 0 being set or cleared (i.e., H, S, V, N, Z, C).
     - WARNING: The Atmel "Instruction Set Summary" pages provided as part of each quiz and exam incorrectly classifies compare instructions (`cp, cpc, cpi`) as "Branch Instructions." They should be listed under "Arithmetic and Logical Instructions." To highlight this inconsistency on Atmel's part, the `tst` instruction is correctly listed under "Arithmetic and Logical Instructions."
     - To allow for multiple branch conditions to be tested, these instructions typically do not modify any of our 32 general purpose registers. *For compare instructions, this is accomplished by a subtraction without a destination operand.*
  2. The second instruction is a conditional branch instruction testing one or more SREG flag bits.

CONDITIONAL BRANCH INSTRUCTION SUMMARY

- As mentioned in the previous slide, typically a conditional control transfer instruction follows a compare or test instruction, where some relationship between two registers is being studied. The following table may be used to quickly find the correct conditional branch instructions for these conditions.

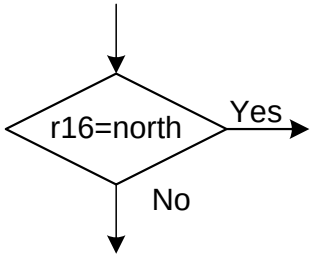
Data Type	Test	SREG bit	Mnemonic	Complementary `If(Test){}`	SREG bit	Mnemonic
Signed	$R_d > R_r$	$R_d > R_r = R_r < R_d$	BRLT[5]	$R_d \leq R_r$	$R_d \leq R_r = R_r \geq R_d$	BRGE[5]
Signed	$R_d \geq R_r$	$S = 0$	BRGE	$R_d < R_r$	$S = 1$	BRLT
Signed	$R_d = R_r$	$Z = 1$	BREQ	$R_d \neq R_r$	$Z = 0$	BRNE
Signed	$R_d \leq R_r$	$R_d \leq R_r = R_r \geq R_d$	BRGE[5]	$R_d > R_r$	$R_d > R_r = R_r < R_d$	BRLT[5]
Signed	$R_d < R_r$	$S = 1$	BRLT	$R_d \geq R_r$	$S = 0$	BRGE
Unsigned	$R_d > R_r$	$R_d > R_r = R_r < R_d$	BRLO[5]	$R_d \leq R_r$	$R_d \leq R_r = R_r \geq R_d$	BRSH[5]
Unsigned	$R_d \geq R_r$	$C = 0$	BRSH/BRCC	$R_d < R_r$	$C = 1$	BRLO/BRCS
Unsigned	$R_d = R_r$	$Z = 1$	BREQ	$R_d \neq R_r$	$Z = 0$	BRNE
Unsigned	$R_d \leq R_r$	$R_d \leq R_r = R_r \geq R_d$	BRSH[5]	$R_d > R_r$	$R_d > R_r = R_r < R_d$	BRLO[5]
Unsigned	$R_d < R_r$	$C = 1$	BRLO/BRCS	$R_d \geq R_r$	$C = 0$	BRSH/BRCC
Simple	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

[5] Interchange Rd and Rr in the operation before the test, i.e., CP Rd,Rr → CP Rr,Rd

A Conditional Control Transfer (Branch) Example

- Here is how a high-level language decision diamond would be implemented in assembly.

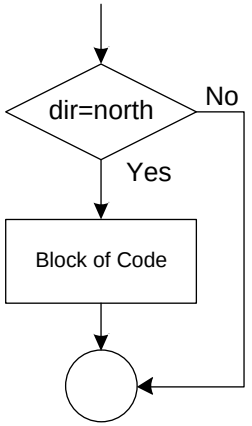
```
; directions (see note)
.EQU south=0b00    ; most significant 6 bits zero
.EQU east=0b01
.EQU west=0b10
.EQU north=0b11

cpi   r16,north    ; step 1: Z flag set if r16 = 0b00000011
breq  yes          ; step 2: branch if Z flag is set
```



Note: These equates are included in `testbench.inc`

IMPLEMENTING A HIGH-LEVEL IF STATEMENT

- A high-level `if` statement is typically comprised of…
  1. Conditional control transfer sequence (last slide) where the complement (not) of the high-level conditional expression is implemented.
  2. High-level procedural block of code is converted to assembly.



- C++ High-level IF Expression

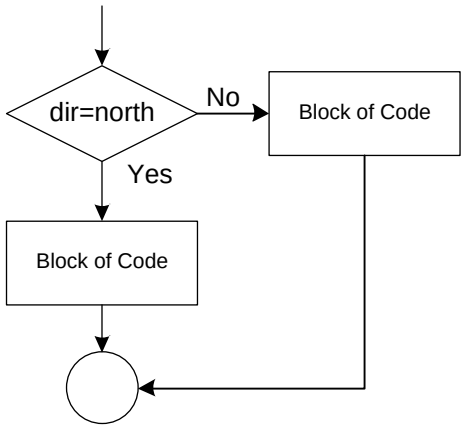
```
if (r16 == north) {
     block of code to be executed if answer is yes.
}
```

- Assembly Version

```
    cpi   r16,north    ; Is bear facing north?
    brne  no           ; branch if Z flag is clear (not equal)
        block of code to be executed if answer is yes.
no:
```

IMPLEMENTING A HIGH-LEVEL IF…ELSE STATEMENT

- A high-level `if…else` statement is typically comprised of…
  1. Conditional control transfer sequence where the complement (not) of the high-level conditional expression is implemented.
  2. High-level procedural block of code for yes (true) condition.
  3. Unconditional jump over the no (false) block of code.
  4. High-level procedural block of code for no (false) condition.



- C++ High-level if…else Expression

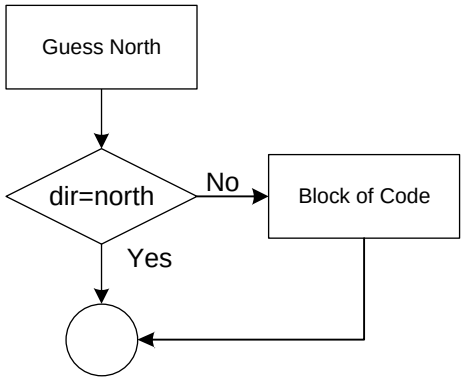
```
if (r16 == north) {
     block of code to be executed if answer is yes (true).
}
else {
     block of code to be executed if answer is no (false).
}
```

- Assembly Version

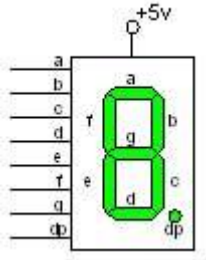
```
     cpi   r16,north    ; Is bear facing north?
     brne  else         ; branch if Z flag is clear (not equal)
          block of code to be executed if answer is yes.
     rjmp  end_if
else:
          block of code to be executed if answer is no.
end_if:
```

ASSEMBLY OPTIMIZATION OF A HIGH-LEVEL IF…ELSE STATEMENT – ADVANCED TOPIC –

- If the if-else blocks of code can be done in a single line of assembly then the program flow is modified to guess the most likely outcome of the test.
    - This is possible if the value of a variable (for example the segments of a 7-segment display to be turned on) is the only thing done in each block.
    - This *optimized* program flow will always execute as fast as the normal if..else program flow (if the guess if wrong) and faster if the guess is correct.
    - This implementation is also more compact and often easier to understand.



- Assembly Version




```
; 7-segment display (see note)
.EQU seg_a=0
.EQU seg_b=1
.EQU seg_c=2
…
    ldi   r17,1<<seg_a  ; guess bear is facing north
    cpi   r16,north     ; Is bear facing north?
    breq  done          ; branch if Z flag is clear (not equal)
    block of code to be executed if guess was wrong.
done:
```

Note: These equates are included in `spi_shield.inc`

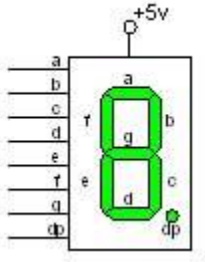
Program Examples

Group A or B – Pseudocode example

- Objective

  Assign the least significant 4 switches on the CSULB shield to group A and the most significant to group B. Based on user input, display A or B based on which group has the higher value. In the event of a tie display E for equal. For this programming problem assume that people choose A 50% of the time, B 40% of the time, and set the switches equal to each other 10% of the time.

- Pseudocode
  - Using the `ReadSwitches` subroutine or reading the I/O ports directly, input group A into register A (`.DEF regA = r16`) and group B into register B (`.DEF regB = r17`)
  - Preload the output register (`.DEF answer = r18`) with the letter `A` ← Guess
  - If (A>B) then go to display answer.
  - Preload the output register with the letter `B` ← Guess
  - If (B>A) then go to display answer.
  - Set answer to E and display answer.



- Seven segment display values.

```
   dpgfedcba
A = 01110111    .SET groupA = 0x77        alternate: .EQU
b = 01111100    .SET groupB = 0x7C
E = 01111001    .SET equal  = 0x79
```

- Programming work around by interchanging Rd and Rr.

			Interchange of A and B		
Solution	Test	Mnemonic	Test	Mnemonic	
Guess	A > B	~~BRLO~~[1]	B < A	BRLO/BRCS	Unsigned

Direction Finder – Two Program Solutions

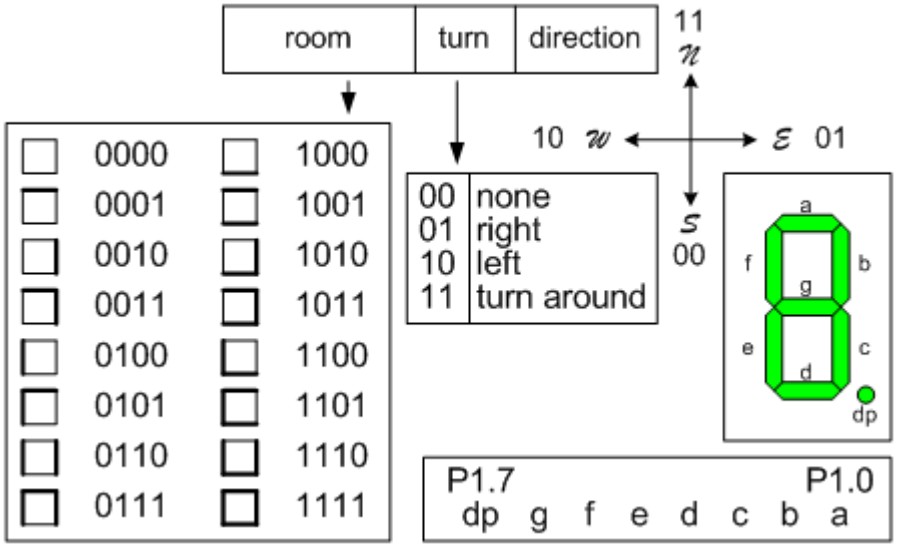
- Objective

  Design a digital circuit with two (2) switches that will turn on one of the rooms 4 LED segments indicating the direction you want your bear to walk

- Direction to Segment Conversion Table

			Inputs		Outputs			
Direction	SW1	SW0	dir.1	dir.0	Direction Segment ON = 1, OFF = 0			
					LDg	LDb	LDf	LDa
South	DWN	DWN	0	0	1	0	0	0
East	DWN	UP	0	1	0	1	0	0
West	UP	DWN	1	0	0	0	1	0
North	UP	UP	1	1	0	0	0	1

- Programmer's Reference Card



Direction Finder – Truth Table Implementation

```
lds   r16, dir       // move direction bits into a working register

// facing east (segment b)
bst   r16,0          // store direction bit 0 into T
bld   var_B,0        // load r16 bit 0 from T
bst   r16,1          // store direction bit 1 into T
bld   var_A,0        // load r17 bit 0 from T
com   var_A          // B = /A * B
and   var_B, var_A
bst   var_B,0        // store r16 bit 0 into T
bld   spi7SEG, seg_b // load r8 bit 1 from T
```

Implementation of Boolean expressions for segments a, f, and g (circuit schematic)

Direction Finder – Using Conditional Expressions

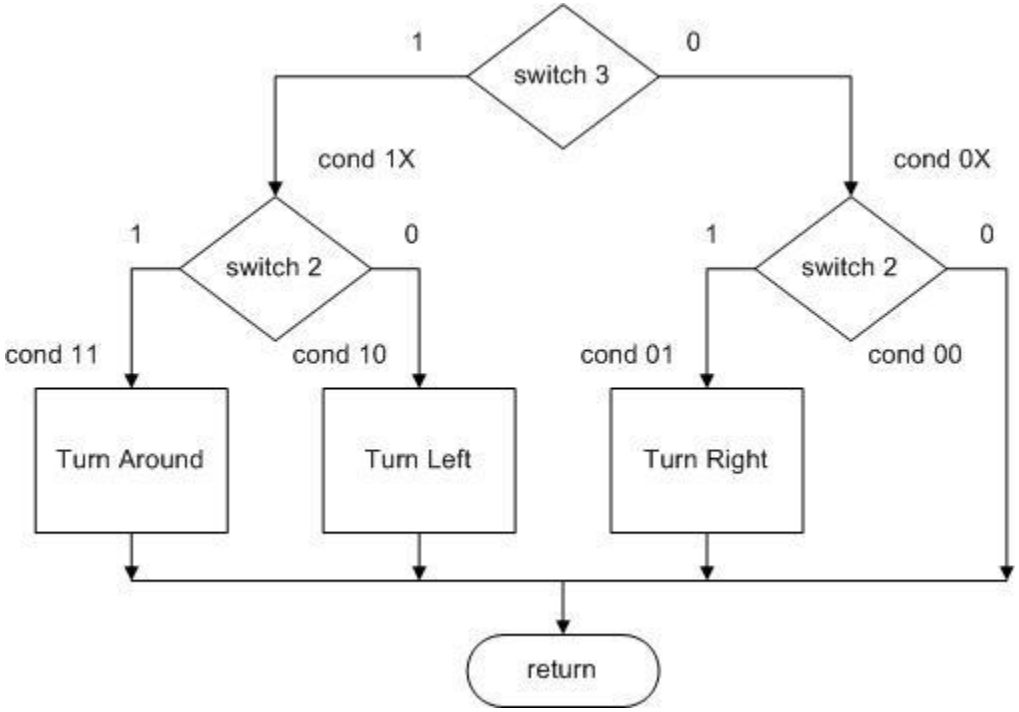
```
    lds   r16, dir

    ldi   r17, 1<<seg_g     ; guess bear is facing south
    cpi   r16,south         ; if bear is facing south then we are done
    breq  done
    ldi   r17, 1<<seg_f     ; guess bear is facing west
    cpi   r16,west          ; if bear is facing west then we are done
    breq  done
    ldi   r17, 1<<seg_b     ; guess bear is facing east
    cpi   r16,east          ; if bear is facing east then we are done
    breq  done
    ldi   r17, 1<<seg_a     ; bear is facing north

done:
    mov   spi7SEG, r17      ; answer to 7-segment register
    call  WriteDisplay
```

Pseudo-Instructions TurnLeft, TurnRight, and TurnAround

Using switches 3 and 2, located on Port C pins 3 and 2 respectively, input an action you want the bear to take. The three possible actions are do nothing, turnLeft, turnRight, and turnAround. Write a subroutine named WhichWay to take the correct action as defined by the following table.

SW.3	SW.2	Action
DWN = 0	DWN = 0	Show direction
DWN = 0	UP = 1	rcall turnRight
UP = 1	DWN = 0	rcall turnLeft
UP = 1	UP = 1	rcall turnAround

Table 5.2 Truth Table of Turn Indicators

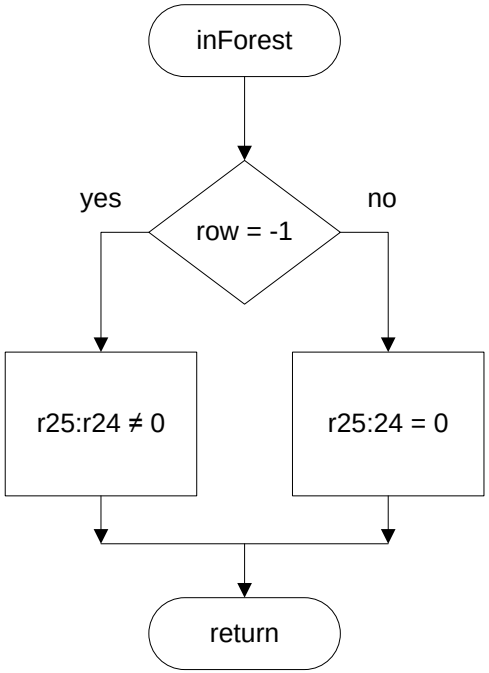


```
; -------------------------
; --- Which Way Do I Go? ---

   call   ReadSwitches  // input port C pins (0x06) into register r7
   bst    switch, 3     // store switch bit 3 into T
   brts   cond_1X       // branch if T is set
   bst    switch, 2     // store switch bit 2 into T
   brts   cond_01       // branch if T is set
cond_00:
   rjmp   whichEnd
cond_01:
   rcall  TurnRight
   rjmp   whichEnd
cond_1X:
   // branch based on the state of switch bit 2
   :
cond_10:
   :
cond_11:
   :
whichEnd:
```

Warning: The above code is for illustrative purposes only and would typically be found in the main looping section of code not in a subroutine. Do not use this code to implement your lab.

InForest and Implementation of IF…ELSE Expression

- The `inForest` subroutine tells us if the bear is in the forest (i.e., has found his way out of the maze).
- The rows and columns of the maze are numbered from 0 to 19 (13h) starting in the upper left hand corner.
- When the bear has found his way out of the maze he is in row minus one (-1). The subroutine is to return true (r25:r24 != 0) if the bear is in the forest and false (r25:r24 == 0) otherwise.
- The register pair r25:r24 is where C++ looks for return values for the BYTE data type.



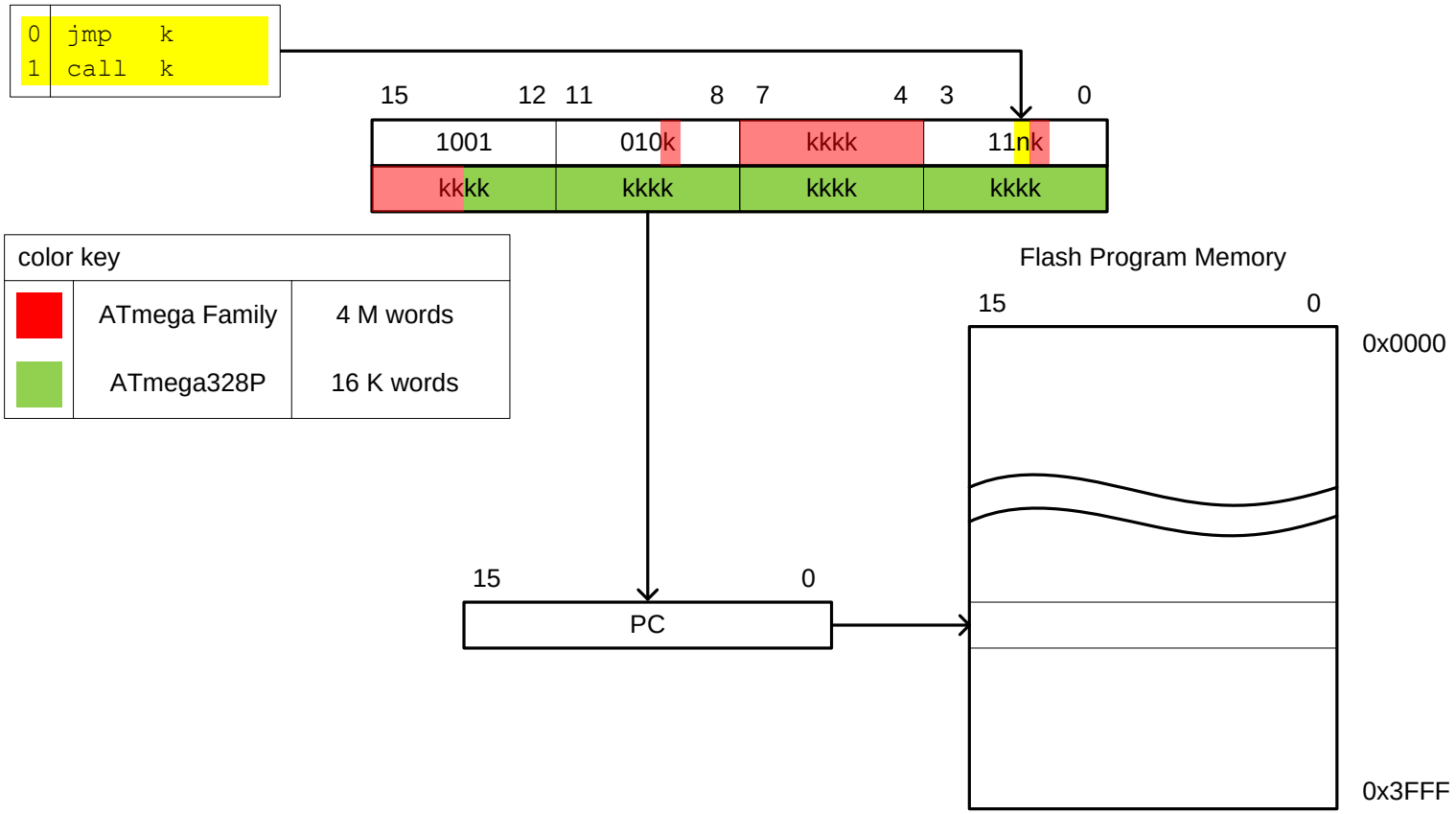
InForest and Implementation of IF…ELSE Expression – Continued –

```
; --------------------------
; ------- In Forest --------
; Called from whichWay subroutine
; Input: row        Outputs: C++ return register (r24)
; No others registers or flags are modified by this subroutine

inForest:
   push   reg_F       // push any flags or registers modified
   in     reg_F,SREG
   push   r16
   lds    r16,row

   test if bear is in the forest

endForest:
   clr    r25         // zero extend
   pop    r16         // pop any flags or registers placed on the stack
   out    SREG,reg_F
   pop    reg_F
   ret
```

APPENDIX A: CONTROL TRANSFER INSTRUCTION ENCODING

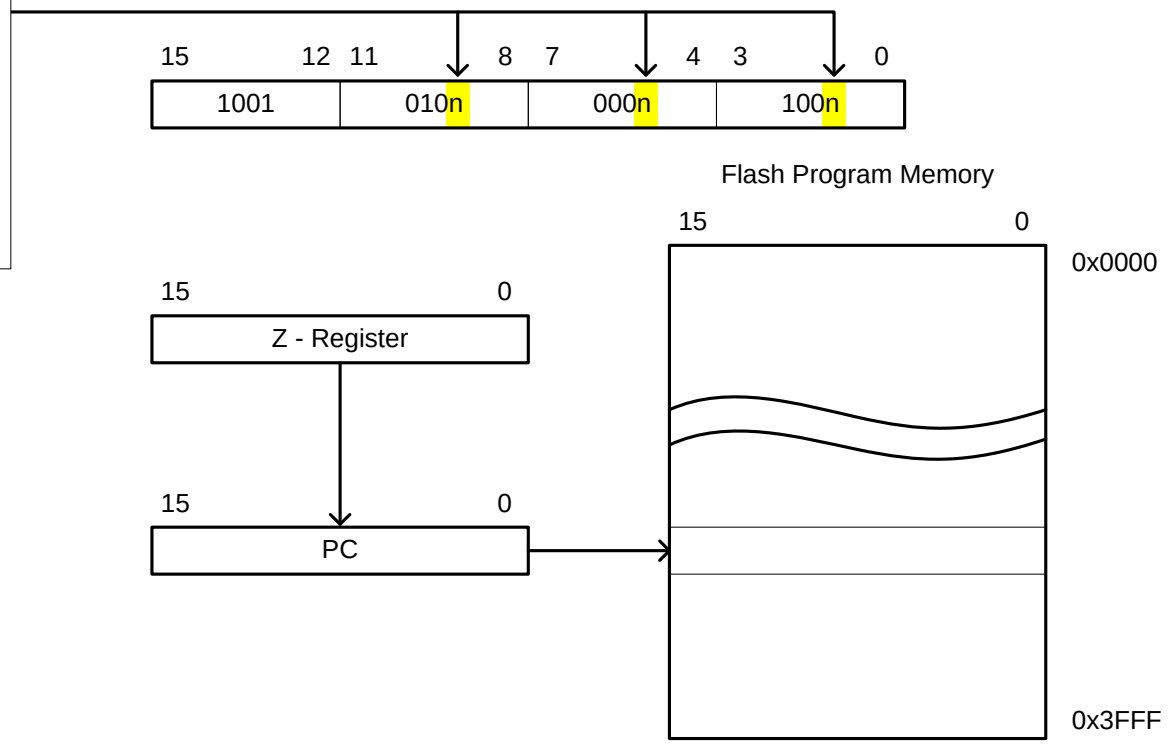
Direct

- All control transfer addressing modes modify the program counter.



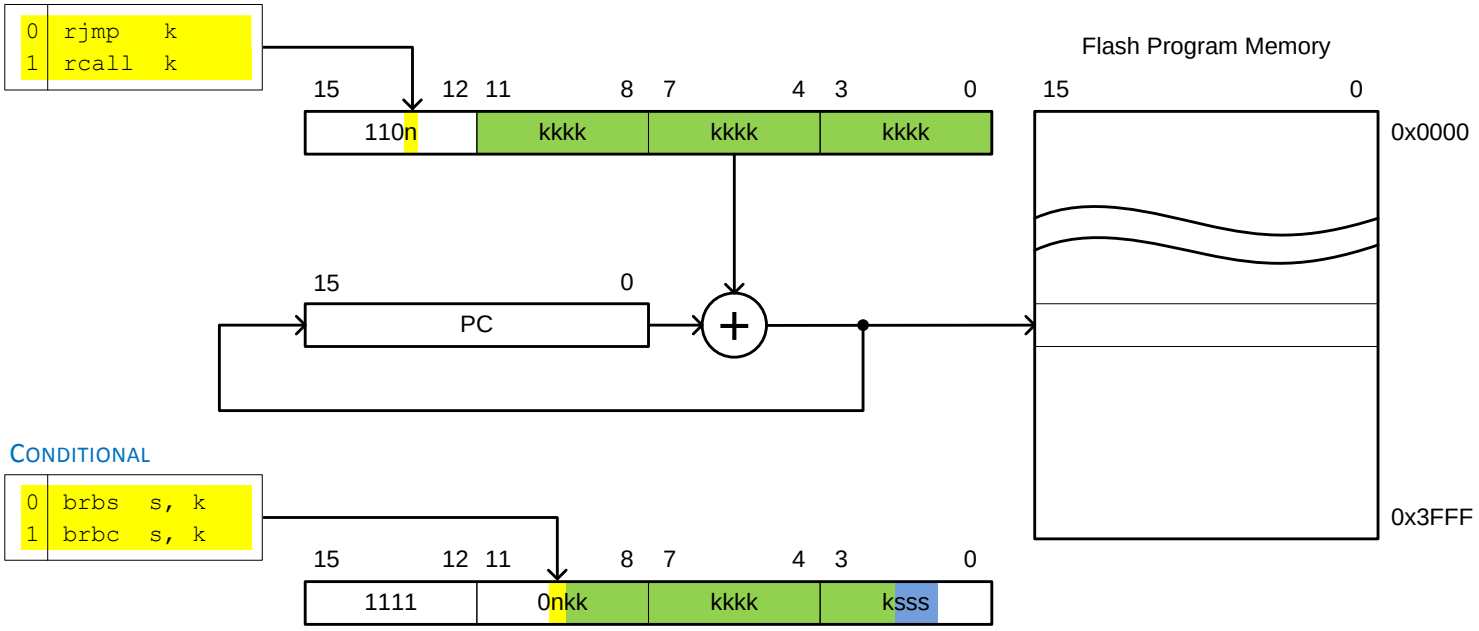
CONTROL TRANSFER INSTRUCTION ENCODING – Indirect

nnn	opcode	notes
000		
001	ijmp	see illustration
010		
011	eijmp	n/a to 328P
100	ret	PC ← M[SP + 1]
101	icall	see illustration
110	reti	PC ← M[SP + 1]
111	eicall	n/a to 328P



CONTROL TRANSFER INSTRUCTION ENCODING – Relative

UNCONDITIONAL



alias			
SREG	sss	brbs s, k	brbc s, k
I	111	brie k	brid k
T	110	brts k	brtc k
H	101	brhs k	brhc k
S	100	brlt k	brge k
V	011	brvs k	brvc k
N	010	brmi k	brpl k
Z	001	breq k	brne k
C	000	brcs k brlo k	brcc k brsh k

NOTES

1. See Register Direct Addressing for encoding of skip register bit set/clear instructions sbrc and sbrs.
2. See I/O Direct Addressing for encoding of skip I/O register bit set/clear instructions sbis and sbic.

APPENDIX B – AVR STATUS REGISTER[6] (SREG)

Bit	7	6	5	4	3	2	1	0	
0x3F (0x5F)	I	T	H	S	V	N	Z	C	SREG
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	

Non ALU

- **Bit 7 – I: Global Interrupt Enable**
  The Global Interrupt Enable bit must be set for the interrupts to be enabled. The individual interrupt enable control is then performed in separate control registers. The I-bit is cleared by hardware after an interrupt has occurred, and is set by the `reti` instruction. The I-bit can also be set and cleared by the application with the `sei` and `cli` instructions.
- **Bit 6 – T: Bit Copy Storage**
  The Bit Copy instructions `bld` (Bit LoaD) and `bst` (Bit STore) use the T-bit as source or destination. A bit from a register can be copied into T ($R_b \rightarrow T$) by the `bst` instruction, and a bit in T can be copied into a bit in a register ($T \rightarrow R_b$) by the `bld` instruction.

ALU

Signed two's complement arithmetic

- **Bit 4 – S: Sign Bit, S = N $\oplus$ V**
  Bit set if answer is negative with no errors or if both numbers were negative and error occurred, zero otherwise.
- **Bit 3 – V: Two's Complement Overflow Flag**
  Bit set if error occurred as the result of an arithmetic operation, zero otherwise.
- **Bit 2 – N: Negative Flag**
  Bit set if result is negative, zero otherwise.

Unsigned arithmetic

- **Bit 5 – H: Half Carry Flag**
  Carry from least significant nibble to most significant nibble. Half Carry is useful in BCD arithmetic.
- **Bit 0 – C: Carry Flag**
  The Carry Flag C indicates a carry in an arithmetic operation. Bit set if error occurred as the result of an unsigned arithmetic operation, zero otherwise.

Arithmetic and Logical

- **Bit 1 – Z: Zero Flag**
  The Zero Flag Z indicates a zero result in an arithmetic or logic operation.

[6] Source: ATmega328P Data Sheet http://www.atmel.com/dyn/resources/prod_documents/8161S.pdf Section 6.3 Status Register

APPENDIX C – CONTROL TRANSFER (BRANCH) INSTRUCTIONS

Compare and Test `cp, cpc, cpi, tst, bst`

Unconditional

- Relative (1) `rjmp, rcall`
- Direct `jmp, call`
- Indirect `ijmp, icall`
- Subr. & Inter. Return `ret, reti`

Conditional

- Branch if (2) …
  - SREG Flag bit is clear (`br`*`Flag`*`c`) or set (`br`*`Flag`*`s`) by name (I, **T**, **H**, S, **V**, N, Z, **C**) or bit (`brbc`, `brbs`).
  - These SREG flag bits (I, T, H, **S**, V, **N**, **Z**, **C**) use more descriptive mnemonics.
    - ✓ Branch if equal (`breq`) or not equal (`brne`) test the Z flag.
    - ✓ Unsigned arithmetic branch if plus (`brpl`) or minus (`brmi`) test the N flag, while branch if same or higher (`brsh`) or lower (`brlo`), test the C flag and are equivalent to `brcc` and `brcs` respectively.
    - ✓ Signed 2's complement arithmetic branch if number is less than zero (`brlt`) or greater than or equal to zero (`brge`) test the S flag
- Skip if …
  - Bit (b) in a register is clear (`sbrc`) or set (`sbrs`).
  - Bit (b) in I/O register is clear (`sbic`) or set (`sbis`). Limited to I/O addresses 0-31

Note:

1. Branch relative to PC + ($-2^{k-1} \Rightarrow 2^{k-1}-1$, where k = 12) + 1 ⇨PC-2047 to PC+2048, within 16 K word address space of ATmega328P
2. All branch relative to PC + ($-2^{k-1} \Rightarrow 2^{k-1}-1$, where k = 7) + 1 ⇨PC-64 to PC+63, within 16 K word address space of ATmega328P

APPENDIX D – ATMEGA328P INSTRUCTION SET[7]

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

[7] 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.