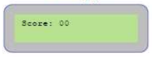
Problem #3 [25 pts]
Consider the following AVR assembly code from your lab that modifies the TekBot to push an object. Some modifications to the wiring of the TekBot are made as shown in the diagram on the next page.

```
            .def    waitcnt = r17
            .equ    WTime = ____
            .org    $0000
            rjmp    INIT
            .org    $0046

 INIT:      ldi     r16, low(RAMEND)
            out     SPL, r16
            ldi     r16, high(RAMEND)
            out     SPH, r16
            ___     ____________            ; Initialize Port A for output
            ___     ____________
            ldi     r16, $00                ; Set Port A outputs to low
            out     PORTA, r16
            ___     ____________            ; Initialize Port B for input
            ___     ____________
            ___     ____________            ; Activate pull-up resistors
            ___     ____________            ; to set up active low pins
            (… Set TekBot to Move Forward: one line …)
            out     PORTA, r16
 MAIN:      in      r16, PINB               ; read whisker inputs
            (… mask and check if right whisker
             … is hit - 2 lines of code …)
            brne    NEXT                    ; if right whisker not hit branch
            rcall   HitRight                ; otherwise run HitRight subroutine
            rjmp    MAIN
 NEXT:      (… check if left whisker hit: one line …)
            brne    MAIN                    ; if no whisker hit continue with loop
            rcall   HitLeft                 ; run HitLeft subroutine
            rjmp    MAIN

 HitRight:  ldi     r16, $00                ; Set output port to move backwards
            out     PORTA, r16
            ldi     waitcnt, WTime
            rcall   Wait
            ___     ____________            ; Turn right
            out     PORTA, r16
            ldi     waitcnt, WTime
            rcall   Wait
            (… Move forward again: one line…)
            out     PORTA, r16
            ret

 HitLeft:   ldi     r16, $00                ; Set output port to move backwards
            out     PORTA, r16
            ldi     waitcnt, WTime
            rcall   Wait
            ___     ____________            ; Turn left
            out     PORTA, r16
            ldi     waitcnt, WTime
            rcall   Wait
            (… Move forward again: one line …)
            out     PORTA, r16
            ret
```

Problem #4 [25 pts]
Assume one of the ATmega128 external interrupts is wired to a push-button to tally a score. Each time the button is pressed the score is incremented by two and displayed on line 1 of the LCD display. The AVR code below (with some missing information) implements this system. Recall from lab that the LCDDriver subroutines use data memory locations $0100-$010F for line 1 of the display. Also, LCDLn1Addr = $0100.

[4 pts] (a) Which external interrupt is being used to trigger the score update and what Port is it on?
[3 pts] (b) Is the interrupt configured to trigger on a rising edge, falling edge, or low level? Explain.
[8 pts] (c) Fill in the lines labeled (1), (2), (3), (4) for Z and Y pointer initialization.
[4 pts] (d) Write the code necessary in lines (5) and (6) to load the characters stored at the SCORE label and store them into the data memory locations reserved for line 1 of the LCD display.
[2 pts] (e) Fill line (7) with the appropriate immediate value.
[4 pts] (f) Based on the description of the BIN2ASCII and ASCII2BIN subroutines, fill in the proper register values for the ld and st instructions on lines (8), (9), (10), and (11)?

LCD Display



```
            .include "m128def.inc"
            .def    mpr = r16
            .def    length = r17
            .def    two = r18
            .org    $0000
START:      rmp     INIT
            .org    $0010
            rcall   UPDATE
            reti
            .org    $0046
INIT:       __________(1)__________     ; Initialize Z pointer
            __________(2)__________
            __________(3)__________     ; Initialize Y pointer
            __________(4)__________
            ldi     length, ___(7)___
            ldi     two, 2

LOAD:       __________(5)__________     ; Load 8-bit data from prog memory
            __________(6)__________     ; Store data to data memory
            dec     length
            brne    LOAD

            ldi     mpr, 0b11000000
            out     EICRB, mpr
            ldi     mpr, 0b10000000
            out     EIMSK, mpr
            ldi     mpr, $00
            out     DDRE, mpr
            ldi     mpr, $FF
            out     PORTE, mpr
            sei

MAIN:       rcall   LCDWrLn1
            rjmp    MAIN

UPDATE:     ldi     YL, low($107)       ; initialize Y pointer to point to first
            ldi     YH, high($107)      ; digit of score (high byte)
            ld      ___(8)___, Y+       ; read in ASCII representation
            ld      ___(9)___, Y
            rcall   ASCII2BIN           ; Convert to binary
            ADD     r0, two             ; Update score value by adding 2
            rcall   BIN2ASCII           ; Convert to ASCII
            st      Y, ___(10)___       ; write new ASCII score to data memory
            st      -Y, ___(11)___
            ret
BIN2ASCII:  {…
            ; read binary value located in r0, writes ASCII conversion to r1:r0
            ret
          }
ASCII2BIN:  {…

            ; read ascii value into r1:r0, write binary conversion to r0
            ret
          }
SCORE:      .DB "Score: 00        "         ; sixteen characters
            .include "LCDDriver.asm"
```

(a)

Looking at the EIMSK register we see that the most significant bit is set. Another way of knowing the interrupt is by looking at the Interrupt vector being used, which is $0010. Therefore, **External interrupt 7 or (INT7)** is being used to trigger the score update.

This interrupt is on **Port E**.

(b) The interrupt is configured as **rising edge**

We know this from how the External Interrupt Control Register B (EICRB) is being set. Notice that the two most significant bits which correspond to Interrupt 7 and are set to 11 which configure the interrupt sensors to be rising edge.

HitRight Subroutine

```
;-----------------------------------------------------------------
; Sub:    HitRight
; Desc:   Handles functionality of the TekBot when the right whisker
;         is triggered.
;-----------------------------------------------------------------
HitRight:
;;Save mpr, wait, SREG registers
; Move Backwards for a second
     ldi       r16, $00           ; Load Move Backwards command
     out       PORTB, r16         ; Send command to port
     ldi       waitcnt, Wtime     ; Wait for 1 second (WTime=100, waitcnt=r17)
     rcall     Wait               ; Call wait function
; Turn left for a second
     ldi       r16, 0b00100000    ; Load Turn Left Command (PORTB, pin 6)
     out       PORTB, r16         ; Send command to port
     ldi       waitcnt, WTime     ; Wait for 1 second
     rcall     Wait               ; Call wait function
; Move Forward again
     ldi       r16, 0b01100000    ; Load Move Forward command
     out       PORTB, r16         ; Send command to port
;;Restore mpr, wait, SREG registers
     ret                          ; Return from subroutine
```

Wait Subroutine

16 MHz clock rate => 62.5 nsec per clock cycle
Why triple-nested loop? Only 8-bit registers!

```
;-----------------------------------------------------------------
; Sub:    Wait
; Desc:   A wait loop that is 16 + 159975*waitcnt cycles or roughly
;         waitcnt*10ms.  Just initialize wait for the specific amount
;         of time in 10ms intervals. Here is the general equation
;         for the number of clock cycles in the wait loop:
;             ((3 * ilcnt + 3) * olcnt + 3) * waitcnt + 13 + call
;-----------------------------------------------------------------
Wait:
;; Save wait, r18, r19 registers
Loop:
     ldi  r19, 224  (1)       ; load outer-loop counter register
OLoop:
     ldi  r18, 237  (1)       ; load inner-loop counter register
ILoop:
     dec  r18       (1)       ; decrement ilcnt
     brne Iloop     (2/1)     ; Continue Inner Loop
     dec  r19       (1)       ; decrement olcnt
     brne Oloop     (2/1)     ; Continue Outer Loop
     dec  waitcnt   (1)       ; Decrement wait
     brne Loop      (2/1)     ; Continue Wait loop
;; restore wait, r18, r19 registers
     ret                      ; Return from subroutine
```

Interrupt Vectors

```
;***********************************************************
;*    Start of Code Segment
;***********************************************************
      .cseg                   ; Beginning of code segment

;-----------------------------------------------------------
; Interrupt Vectors
;-----------------------------------------------------------
      .org     $0000          ; Beginning of IVs
      rjmp     INIT           ; Reset interrupt

      .org     $000A {IRQ4 => pin4, PORTE}
      rcall    HitRight       ; Call hit right function
      reti                    ; Return from interrupt
      .org     $000C {IRQ5 => pin5, PORTE}
      rcall    HitLeft        ; Call hit left function
      reti                    ; Return from interrupt

      .org     $0046          ; End of Interrupt Vectors
```

Initialization

```
;-----------------------------------------------------------
; Program Initialization
;-----------------------------------------------------------
INIT: ; The initialization routine
      ; Initialize Stack Pointer
      ldi   mpr, high(RAMEND)
      out   SPH, mpr
      ldi   mpr, low(RAMEND)
      out   SPL, mpr
      ; Initialize Port B for output
      ldi   mpr, (1<<EngEnL)|(1<<EngEnR)|(1<<EngDirR)|(1<<EngDirL)
      out   DDRB, mpr        ; Set the DDR register for Port B
      ldi   mpr, $00
      out   PORTB, mpr       ; Set the default output for Port B
      ; Initialize Port E for input
      ldi   mpr, (0<<WskrL)|(0<<WskrR)
      out   DDRE, mpr        ; Set the DDR register for Port E
      ldi   mpr, (1<<WskrL)|(1<<WskrR)
      out   PORTE, mpr       ; Set the Port E to Input with Hi-Z
      ; Initialize external interrupts
      ; Set the Interrupt Sense Control to falling edge
      ldi   mpr, (1<<ISC41)|(0<<ISC40)|(1<<ISC51)|(0<<ISC50)
      out   EICRB, mpr       ; Set INT4 & 5 to trigger on falling edge
      ldi   mpr, $00
      sts   EICRA, mpr       ; Use sts, EICRA in extended I/O space
      ; Set the External Interrupt Mask
      ldi   mpr, (1<<INT4)|(1<<INT5)
      out   EIMSK, mpr
      ; Turn on interrupts
      sei
```

Initialization: Ports

```
; Initialize Port B for output

              Bit 7        Bit 4        Bit 5         Bit 6
ldi   mpr, (1<<EngEnL)|(1<<EngEnR)|(1<<EngDirR)|(1<<EngDirL) DDRB = 11110000 for output
out   DDRB, mpr                    ; Set the DDR register for Port B
ldi   mpr, $00
out   PORTB, mpr                   ; Set the default output for Port B

; Initialize Port E for input
             Bit 5      Bit 4
ldi   mpr, (0<<WskrL)|(0<<WskrR) DDRE = 00000000 for input
out   DDRE, mpr                    ; Set the DDR register for Port E
ldi   mpr, (1<<WskrL)|(1<<WskrR)
out   PORTE, mpr                   ; Set the Port E to Input with Hi-Z
```

Initialization: Interrupts

```
 ; Set the Port E to Input with Hi-Z
 ; Initialize external interrupts
 ; Set the Interrupt Sense Control to low level
 ldi   mpr, (1<<ISC41)|(0<<ISC40)|(1<<ISC51)|(0<<ISC50) EICRB = 00001010
 out   EICRB, mpr
 ldi   mpr, $00
 sts   EICRA, mpr
 ; Set the External Interrupt Mask
 ldi   mpr, (1<<INT4)|(1<<INT5) EIMSK = 00110000
 out   EIMSK, mpr
 ; Turn on interrupts
 sei
```

External Interrupt Control Register B (EICRB)

ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40
				1	0	1	0

Set to trigger on falling edge

External Interrupt Mask Register (EIMSK)

INT7	INT6	INT5	INT4	INT3	INT2	INT1	INT0
		1	1	=> Enabled			

HitRight Subroutine

```
;-----------------------------------------------------------------
; Sub: HitRight
; Desc:      Handles functionality of the TekBot when the right whisker
;            is triggered.
;-----------------------------------------------------------------
HitRight:
      push   mpr                                ; Save mpr register
      push   waitcnt                ; Save wait register
      in     mpr, SREG              ; Save program state
      push   mpr                                ;

      ; Move Backwards for a second
      ldi    mpr, MovBck            ; Load Move Backwards command
      out    PORTB, mpr             ; Send command to port
      ldi    waitcnt, WTime                     ; Wait for 1 second
      rcall Wait                                ; Call wait function

      ; Turn left for a second
      ldi    mpr, TurnL             ; Load Turn Left Command
      out    PORTB, mpr             ; Send command to port
      ldi    waitcnt, WTime                     ; Wait for 1 second
      rcall Wait                                ; Call wait function

      pop    mpr                                ; Restore program state
      out    SREG, mpr              ;
      pop    waitcnt                ; Restore wait register
      pop    mpr                                ; Restore mpr
      ret                                       ; Return from subroutine
```

```
ADD16:
            ; Save variable by pushing them to the stack
            push    A                           ; Save A register
            push    B                           ; Save B Register
            push    zero                ;
            push    XH                          ; Save X-ptr
            push    XL
            push    YH                          ; Save Y-ptr
            push    YL
            push    ZH
            push    ZL

            clr             zero
            ; Execute the function here

            ldi             YL, low(addrB) ; Load low byte
            ldi             YH, high(addrB); Load high byte

            ldi             XL, low(addrA) ; Load low byte
            ldi             XH, high(addrA); Load high byte

            ldi             ZL, low(addrSL)
            ldi             ZH,     high(addrSH)

            ld              A, X+   ;Get byte of A
            ld              B, Y+   ; Get byte of B
            add             A, B    ; add them together
            st              Z+, A   ; Store into Z
            ld              A, X    ; Get second byte of A
            ld              B, Y    ; Get second byte of B
            adc             A, B    ; Add with carry A and B
            st              Z+, A   ; Store A into Z
            clr A
            adc A, zero
            st Z, A

            ; Restore variable by popping them from the stack in reverse order\
            pop ZL
            pop ZH
            pop YL
            pop YH
            pop XL
            pop XH
            pop zero
            pop B
            pop A

            ret                                  ; End a function with RET
```

```
;-----------------------------------------------------------------
MUL24:
           ; Save variable by pushing them to the stack
           push   A                           ; Save A register
           push   B                           ; Save B register
           push   rhi                         ; Save rhi register
           push   rlo                         ; Save rlo register
           push   zero                  ; Save zero register
           push   XH                          ; Save X-ptr
           push   XL
           push   YH                          ; Save Y-ptr
           push   YL
           push   ZH                          ; Save Z-ptr
           push   ZL
           push   oloop                 ; Save counters
           push   iloop

           clr          zero                  ; Maintain zero semantics


                        ; Set Y to beginning address of B
           ldi          YL, low(addrSL)       ; Load low byte
           ldi          YH, high(addrSH)      ; Load high byte

           ; Set Z to begginning address of resulting Product
           ldi          ZL, low(LAddrP)       ; Load low byte
           ldi          ZH, high(LAddrP); Load high byte

           ; Begin outer for loop
           ldi          oloop, 3              ; Load counter
MUL24_OLOOP:
           ; Set X to beginning address of A
           ldi          XL, low(addrSL)       ; Load low byte
           ldi          XH, high(addrSL)      ; Load high byte

           ; Begin inner for loop
           ldi          iloop, 3              ; Load counter


MUL24_ILOOP:
         ld           A, X+                ; Get byte of A operand
         ld           B, Y                 ; Get byte of B operand
         mul          A,B                  ; Multiply A and B
         ld           A, Z+                ; Get a result byte from memory
         ld           B, Z+                ; Get the next result byte from memory
         add          rlo, A               ; rlo <= rlo + A
         adc          rhi, B               ; rhi <= rhi + B + carry
         ld           A, Z                 ; Get a third byte from the result
         adc          A, zero              ; Add carry to A
         st           Z, A                 ; Store third byte to memory
         st           -Z, rhi              ; Store second byte to memory
         st           -Z, rlo              ; Store third byte to memory
         adiw   ZH:ZL, 1            ; Z <= Z + 1
         dec          iloop                ; Decrement counter
         brne   MUL24_ILOOP         ; Loop if iLoop != 0
         ; End of inner for Loop

         sbiw   ZH:ZL, 2            ; Z <= Z - 1
         adiw   YH:YL, 1            ; Y <= Y + 1
         dec          oloop                ; Decrement counter
         brne   MUL24_OLOOP         ; Loop if oLoop != 0

         ; end of Outer for loop

         ; Restore variable by popping them from the stack in reverse order\
         pop          iloop                ; Restore all registers in reverves order
         pop          oloop
         pop          ZL
         pop          ZH
         pop          YL
         pop          YH
         pop          XL
         pop          XH
         pop          zero
         pop          rlo
```

```
 ,
 INIT:
        ;Stack Pointer (VERY IMPORTANT!!!!)
        ldi mpr, LOW(RAMEND)
        out SPL, mpr
        ldi mpr, HIGH(RAMEND)
        out SPH, mpr

 ;I/O Ports
        ; Initialize Port B for output
        ; Lights on Receiver board
        ldi     mpr, $00                  ; set PORTB to output
        out     PORTB, mpr
        ldi     mpr, $FF                  ; set Port B directional register
        out DDRB, mpr                     ; for output

        ; Initialize Port D for input
        ldi     mpr, $FF                  ; set PortD for inputs
        out PORTD, mpr
        ldi     mpr, $00                  ; set Port D directional register for inputs
        out DDRD, mpr                     ; for input

 ;USART1
        ;Set to Normal data rate
        ; baudrate at 2400bps
        ldi     mpr, $01
        sts     UBRR1H, mpr
        ldi mpr, $A0
        sts UBRR1L, mpr

        ;Set baudrate at 2400bps
        ;ldi mpr, high(416)
        ;sts UBRR1H, mpr
        ;ldi mpr, low(416)
        ;sts    UBRR1L, mpr

        ;Enable receiver and enable receive interrupts
        ldi mpr, (1<<RXEN1)|(1<<RXCIE1)|(1<<TXEN1)
        sts UCSR1B, mpr

        ;Set frame format: 8data bits, 2 stop bit, synchronous
        ldi mpr, (1<<USBS1)|(3<<UCSZ10)
        sts     UCSR1C, mpr


;External Interrupts
        ;Set the External Interrupt Mask
        ldi     mpr, (1<<INT1)|(1<<INT0)
        out     EIMSK, mpr

        ;Set the Interrupt Sense Control to Rising Edge detection
        ldi     mpr, (1<<ISC00)|(1<<ISC01)|(1<<ISC10)|(1<<ISC11)
        sts     EICRA, mpr             ; use sts, EICRA in extended I/O space

        ldi     mpr, $00
        sts     EICRB, mpr

        ;Other
        ldi     hit_cnt, 3             ;start hit count at 3 so ready to decrement

        sei
```

[20 pts]
1- (a) Convert 369.3125_{10} to binary, octal, and hexadecimal.
(b) Convert 10111101.101_2 to decimal, octal, and hexadecimal.
(c) Convert 326.5_8 to decimal, binary, and hexadecimal.
(d) Convert $F3C7.A_{16}$ into binary, octal, and decimal.
Show the entire conversion process.

(a) 369.3125_{10} consists of a integer part (369) and a fraction part (0.3125).
Fraction = 0.3125

Integer

$0.3125 \times 2 = 0.625 \Rightarrow 0$

$0.625 \times 2 = 1.25 \Rightarrow 1$

$0.25 \times 2 = 0.5 \Rightarrow 0$

$0.5 \times 2 = 1.0 \Rightarrow 1$

$0.3125_{10} = 0.0101_2$

Thus, $369.3125_{10} = 101110001.0101_2$

In octal => 561.24_8

In hexidecimal => 171.5_{16}

(c) To obtain decimal

$326.5_8 = 3\times8^2 + 2\times8^1 + 6\times8^0 + 5\times8^{-1} = 192 + 16 + 6 + 0.625 = 214.625_{10}$

To obtain binary

$326.5_8 \Rightarrow 11\ 010\ 110\ .\ 101_2 = 11010110.101_2$

To obtain hexadecimal

$326.5_8 \Rightarrow 11010110.101_2 \Rightarrow 1101\ 0110\ .\ 1010_2 = D6.A_{16}$

(d) There are many ways to do this. To obtain decimal, we do the following

$F3C7.A_{16} \Rightarrow 15\times16^3 + 3\times16^2 + 12\times16^1 + 7\times16^0 + 10\times16^{-1} = 62407.625_{10}$

To obtain binary,

$F3C7.A_{16} = 1111\ 0011\ 1100\ 0111\ .\ 1010_2$

Once we have binary, we can easily convert it to octal
$1\ 111\ 001\ 111\ 000\ 111\ .\ 101\ 0_2 = 171707.5_8$

1- The following binary numbers are signed 8-bit values. Perform the indicated operations in (a) sign-magnitude, (b) 1's-complement, and (c) 2's-complement. Indicate whether or not overflow occurs.

00011011 + 01100100
10011100 + 00011101
11000000 - 10111111
00111001 - 10111000

```
(a)      0  0  0  1  1  0  1  1   (27)
       + 0  1  1  0  0  1  0  0   (100)
         0  1  1  1  1  1  1  1   (127)
   (b) and (c)   =>  same as (a)
```

```
(a)      1  0  0  1  1  1  0  0   (-28)     (b)      1  0  0  1  1  1  0  0   (-99)
       + 0  0  0  1  1  1  0  1   (+29)            + 0  0  0  1  1  1  0  1   (29)
                                                     1  0  1  1  1  0  0  1   (-70)
         0  0  0  1  1  1  0  1   (29)
       - 1  0  0  1  1  1  0  0   (28)

(c)      1  0  0  1  1  1  0  0   (-100)
       + 0  0  0  1  1  1  0  1   (29)
         1  0  1  1  1  0  0  1   (-71)
```

```
(a)      1  1  0  0  0  0  0  0   (-64)     (b)      1  1  0  0  0  0  0  0   (-63)
       - 1  0  1  1  1  1  1  1   (-63)            - 1  0  1  1  1  1  1  1   (-64)
         1  0  0  0  0  0  0  1   (-1)
                                                     1  1  0  0  0  0  0  0   (-63)
                                                   + 0  1  0  0  0  0  0  0   (64)
                                                   1 0  0  0  0  0  0  0  0
                                                   +                      1
                                                     0  0  0  0  0  0  0  1   (1)

(c)      1  1  0  0  0  0  0  0   (-64)
       - 1  0  1  1  1  1  1  1   (-65)

         1  1  0  0  0  0  0  0   (-64)
       + 0  1  0  0  0  0  0  1   (65)
       1 0  0  0  0  0  0  0  1   (1)
```

```
(a)      0  0  1  1  1  0  0  1   (57)      (b)      0  0  1  1  1  0  0  1   (57)
       - 1  0  1  1  1  0  0  0   (-56)            - 1  0  1  1  1  0  0  0   (-71)

         0  0  1  1  1  0  0  1   (57)               0  0  1  1  1  0  0  1   (57)
       + 0  0  1  1  1  0  0  0   (56)             + 0  1  0  0  0  1  1  1   (71)

         0  1  1  1  0  0  0  1   (113)              1  0  0  0  0  0  0  0
                                                     Carry-in, no-carry-out => overflow!

(c)      0  0  1  1  1  0  0  1   (57)
       - 1  0  1  1  1  0  0  0   (-72)

         0  0  1  1  1  0  0  1   (57)
       + 0  1  0  0  0  1  1  1   (72)
         1  0  0  0  0  0  0  0
         Carry-in, no-carry-out => overflow!
```

(b) [10 pts] Multiply the following **2's complement** operands using add and shift implementation

```
(i)      0  1  1  1  0   (Mult)        (ii)     0  1  0  1  1   (Mult)
       × 0  1  0  1  1   (Mpy)                × 1  0  1  0  1   (Mpy)
```

(b)

(i)

```
Mult              C     Acc             Mpy
1  1  1  0        0     0  0  0  0      1  0  1  1
              +   0     1  1  1  0                  Add & shift
                  0     1  1  1  0
                  0     0  1  1  1      0  1  0  1
              +   0     1  1  1  0                  Add & shift
                  1     0  1  0  1
                  0     1  0  1  0      1  0  1  0
                                                    Shift
                  0     0  1  0  1      0  1  0  1
              +   0     1  1  1  0                  Add & shift
                  1     0  0  1  1
                  0   | 1  0  0  1      1  0  1  0 |
                            Final Result = 154
```

(ii)

Since the multiplier is negative, we convert it to positive first before proceeding. The two's complement of 10101 = 01011.

```
Mult              C     ACC             Mpy
1  0  1  1        0     0  0  0  0      1  0  1  1
              +   0     1  0  1  1                  Add & shift
                  0     1  0  1  1
                  0     0  1  0  1      1  1  0  1
              +   0     1  0  1  1                  Add & shift
                  1     0  0  0  0
                  0     1  0  0  0      0  1  1  0
                                                    Shift
                  0     0  1  0  0      0  0  1  1
              +   0     1  0  1  1                  Add & Shift
                  0     1  1  1  1
                  0     0  1  1  1      1  0  0  1
                                  Result = 121
```

This result must be then converted to negative. Therefore, the Final Result is:

```
        | 1  0  0  0      0  1  1  1 |
             Final Result = -121
```

3- Multiply the followings 2's-complement operands using add and shift implementation. Only the machine implementation is acceptable.

```
(i)      0  1  1  1  1   (Mult)        (ii)     0  0  0  1  1   (Mult)
       × 0  1  1  0  1   (Mpy)                × 1  1  1  0  1   (Mpy)
```

(i)

```
Mult              E     Acc             Mpy
1  1  1  1        0     0  0  0  0      1  1  0  1
              +   0     1  1  1  1                  Add & shift
                  0     1  1  1  1
                  0     0  1  1  1      1  1  1  0
                                                    Shift
                  0     0  0  1  1      1  1  1  1
              +   0     1  1  1  1                  Add & shift
                  1     0  0  1  0
                  0     1  0  0  1      0  1  1  1
              +   0     1  1  1  1                  Add & shift
                  1     1  0  0  0
                  0     1  1  0  0      0  0  1  1
                                     Final Result = 195
```

(ii) Since the multiplier is negative, we convert it to positive first before proceeding.

```
Mult              E     ACC             Mpy
0  0  1  1        0     0  0  0  0      0  0  1  1
              +   0     0  0  1  1                  Add & shift
                  0     0  0  1  1
                  0     0  0  0  1      1  0  0  1
              +   0     0  0  1  1                  Add & shift
                  0     0  1  0  0
                  0     0  0  1  0      0  1  0  0
                                                    Shift
                  0     0  0  0  1      0  0  1  0
                                                    Shift
                  0     0  0  0  0      1  0  0  1
                                     Result = 9
```

This result must be then converted to negative. Therefore, the Final Result is -9.

(c) [10 pts] Perform the following operation using both restoring and non-restoring techniques (only the machine implementation is acceptable): 110101 ÷ 111
Note that the numbers are unsigned.

To determine if we will be able to use n/2 bits for the quotient and n/2 bits for the remainder, we subtract the divisor from the higher order bits of the dividend:

```
   0xxx
  110101
-  111
   111 (requires non-existent borrow for msb subtraction, so wrong result, i.e. a negative result)
```

since the subtraction 110-111 yields a negative result, we can use n/2 bits to represent both the quotient and the remainder. You may do this by inspection, noting that 111 > 110, so the first quotient digit is 0.

```
   C     ACC         Dividend     Divisor
   0     1  1  0     1  0  1      111
   1     1  0  1     0  1  X      Shift
+  1     0  0  1                  Subtract
   0     1  1  0     0  1  1      (set q2 = 1)
   1     1  0  0     1  1  X      Shift
+  1     0  0  1                  Subtract
   0     1  0  1     1  1  1      (set q1 = 1)
   1     0  1  1     1  1  X      Shift
+  1     0  0  1                  Subtract
   0     1  0  0
   0   | 1  0  0 | | 1  1  1 |    (set q0=1)
     remainder = 4  quotient = 7
```

which yields the correct answer: 53 ÷ 7 = 7 rem 4

(i) 110101 ÷ 111 Non-Restoring:

For non-restoring division, the results of our subtractions are always positive yielding **the same steps as in restoring division above**. In any division problem, if restore steps are not required during restoring division, then the steps taken to perform the division are no different than non-restoring division.

```
   C     ACC         Dividend     Divisor
   0     1  1  0     1  0  1      111
   1     1  0  1     0  1  X      Shift
+  1     0  0  1                  Subtract
   0     1  1  0     0  1  1      (set q2 = 1)
   1     1  0  0     1  1  X      Shift
+  1     0  0  1                  Subtract
   0     1  0  1     1  1  1      (set q1 = 1)
   1     0  1  1     1  1  X      Shift
+  1     0  0  1                  Subtract
   0     1  0  0
   0   | 1  0  0 | | 1  1  1 |    (set q0=1)
     remainder = 4  quotient = 7
```

[20 pts]

4- Multiply the followings 2's-complement operands using add and shift implementation (only the implementation is acceptable).

```
(i)      0  1  1  1  1   (Mult)        (ii)     0  0  0  1  1   (Mult)
       × 0  1  1  0  1   (Mpy)                × 1  1  1  0  1   (Mpy)
```

(i)

```
Mult              E     Acc             Mpy
1  1  1  1        0     0  0  0  0      1  1  0  1
              +   0     1  1  1  1                  Add & shift
                  0     1  1  1  1
                  0     0  1  1  1      1  1  1  0
                                                    Shift
                  0     0  0  1  1      1  1  1  1
              +   0     1  1  1  1                  Add & shift
                  1     0  0  1  0
                  0     1  0  0  1      0  1  1  1
              +   0     1  1  1  1                  Add & shift
                  1     1  0  0  0
                  0     1  1  0  0      0  0  1  1
                                     Final Result = 195
```

(ii) Since the multiplier is negative, we convert it to positive first before proceeding.

```
Mult              E     ACC             Mpy
0  0  1  1        0     0  0  0  0      0  0  1  1
              +   0     0  0  1  1                  Add & shift
                  0     0  0  1  1
                  0     0  0  0  1      1  0  0  1
              +   0     0  0  1  1                  Add & shift
                  0     0  1  0  0
                  0     0  0  1  0      0  1  0  0
                                                    Shift
                  0     0  0  0  1      0  0  1  0
                                                    Shift
                  0     0  0  0  0      1  0  0  1
                                     Result = 9
```

This result must be then converted to negative. Therefore, the Final Result is -9.

[20 pts]

5- Perform the following operations using both restoring and non-restoring techniques (only implementation is acceptable):

(a) 100011 ÷ 001011
(b) 01011101 ÷ 1001

Note that the numbers are unsigned.

Restoring:

```
   E     ACC                   Dividend              Divisor
   0     0  0  0  0  0  0      1  0  0  0  1  1      0  0  1  0  1  1
   0     0  0  0  0  0  1      0  0  0  1  1  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   1     1  1  0  1  1  0
   0     0  0  0  0  0  1      0  0  0  1  1  0      Restore (set q0=0)
   0     0  0  0  0  1  0      0  0  1  1  0  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   1     1  1  0  1  1  1
   0     0  0  0  0  1  0      0  0  1  1  0  0      Restore (set q0=0)
   0     0  0  0  1  0  0      0  1  1  0  0  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   1     1  1  1  0  0  1
   0     0  0  0  1  0  0      0  1  1  0  0  0      Restore (set q0=0)
   0     0  0  1  0  0  0      1  1  0  0  0  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   1     1  1  1  1  0  1
   0     0  0  1  0  0  0      1  1  0  0  0  0
   0     0  1  0  0  0  1      1  0  0  0  0  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   0     0  0  0  1  1  0
   0     0  0  0  1  1  0      1  0  0  0  0  1      (set q0=1)
   0     0  0  1  1  0  1      0  0  0  0  1  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   0     0  0  0  0  1  0
   0     0  0  0  0  1  0      0  0  0  0  1  1      (set q0=1)
   0     0  0  0  0  1  0      0  0  0  0  1  1      Final result
           remainder=> 2                => 3
```

Non-restoring:

```
   E     ACC                   Dividend              Divisor
   0     0  0  0  0  0  0      1  0  0  0  1  1      0  0  1  0  1  1
   0     0  0  0  0  0  1      0  0  0  1  1  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   1     1  1  0  1  1  0      0  0  0  1  1  0      (set q0=0)
   1     1  0  1  1  0  0      0  0  1  1  0  X      Shift
+  0     0  0  1  0  1  1                            Add
   1     1  1  0  1  1  1      0  0  1  1  0  0      (set q0=0)
   1     1  0  1  1  1  0      0  1  1  0  0  X      Shift
+  0     0  0  1  0  1  1                            Add
   1     1  1  1  0  0  1      0  1  1  0  0  0      (set q0=0)
   1     1  1  0  0  1  0      1  1  0  0  0  X      Shift
+  0     0  0  1  0  1  1                            Add
   1     1  1  1  1  0  1      1  1  0  0  0  0      (set q0=0)
   1     1  1  1  0  1  1      1  0  0  0  0  X      Shift
+  0     0  0  1  0  1  1                            Add
   0     0  0  0  1  1  0      1  0  0  0  0  1      (set q0=1)
   0     0  0  1  1  0  1      0  0  0  0  1  X      Shift
+  1     1  1  0  1  0  1                            Subtract
   0     0  0  0  0  1  0      0  0  0  0  1  1      (set q0=1)
   0     0  0  0  0  1  0      0  0  0  0  1  1      Final result
           remainder=> 2          quotient = > 3
```

(b)

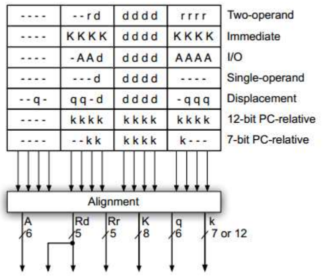
Restoring:

```
   E     ACC              Dividend         Divisor             2's-complement
   0     0  0  0  1  0    1  1  1  0  1    0  1  0  0  1   =>  1  0  1  1  1
   0     0  0  1  0  1    1  1  0  1  X    Shift
+  1     1  0  1  1  1                     Subtract
   1     1  1  1  0  0
   0     0  0  1  0  1    1  1  0  1  0    Restore
   0     0  1  0  1  1    1  0  1  0  X    Shift
+  1     1  0  1  1  1                     Subtract
   0     0  0  0  1  0
   0     0  0  0  1  0    1  0  1  0  1    Restore
   0     0  0  1  0  1    0  1  0  1  X    Shift
+  1     1  0  1  1  1                     Subtract
   1     1  1  1  0  0
   0     0  0  1  0  1    0  1  0  1  0    Restore
   0     0  1  0  1  0    1  0  1  0  X    Shift
+  1     1  0  1  1  1                     Subtract
   0     0  0  0  0  1    1  0  1  0  1
   0     0  0  0  1  1    0  1  0  1  X    Shift
+  1     1  0  1  1  1                     Subtract
   1     1  1  0  1  0    0  1  0  1  0
   0     0  0  0  1  1    0  1  0  1  0    Restore
         remainder=> 3    quotient = > 10
```

Non-Restoring:

```
   E     ACC              Dividend         Divisor             2's-complement
   0     0  0  0  1  0    1  1  1  0  1    0  1  0  0  1   =>  1  0  1  1  1
   0     0  0  1  0  1    1  1  0  1  X    Shift
+  1     1  0  1  1  1                     Subtract
   1     1  1  1  0  0    1  1  0  1  0
   1     1  1  0  0  1    1  0  1  0  X    Shift
+  0     0  1  0  0  1                     Add
   0     0  0  0  1  0    1  0  1  0  1
   0     0  0  1  0  1    0  1  0  1  X    Shift
+  1     1  0  1  1  1                     Subtract
   1     1  1  1  0  0    0  1  0  1  0
   1     1  1  0  0  0    1  0  1  0  X    Shift
+  0     0  1  0  0  1                     Add
   0     0  0  0  0  1    1  0  1  0  1
   0     0  0  0  1  1    0  1  0  1  X    Shift
+  1     1  0  1  1  1                     Subtract
   1     1  1  0  1  0    0  1  0  1  0
   0     0  0  0  1  1    0  1  0  1  0    Restore
         remainder=> 3    quotient = > 10
```

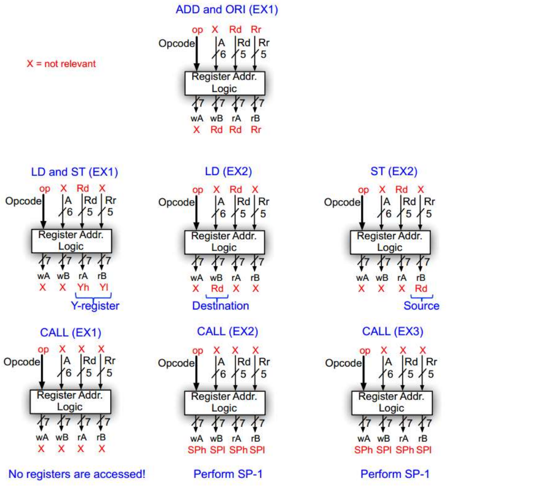
MICRO-ARCHITECTURE

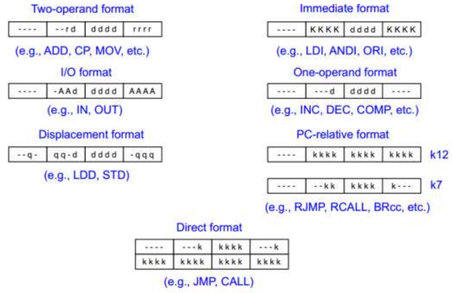


- - - -	- - r d	d d d d	r r r r	Two-operand
- - - -	K K K K	d d d d	K K K K	Immediate
- - - -	- A A d	d d d d	A A A A	I/O
- - - -	- - - d	d d d d	- - - -	Single-operand
- - q -	q q - d	d d d d	- q q q	Displacement
- - - -	k k k k	k k k k	k k k k	12-bit PC-relative
- - - -	- - k k	k k k k	k - - -	7-bit PC-relative

Representative AVR Instructions for Control Unit Design					# of Execute cycles
Category	Mnemonics	Description	Operation	Flags	
ALU	ADD Rd, Rr	Add two Registers	Rd← Rd + Rr	Z,C,N,V,H	1
	ORI Rd, K	Logical OR Register and Constant	Rd← Rd ∨ K	Z,C,N,V,H	1
Data Transfer	LD Rd, Y	Load Indirect	Rd ← M[Y]	None	2
	ST Y, Rr	Store Indirect	M[Y] ← Rr	None	2
Branch	BREQ k	Branch if Equal	if (Z=1) then PC←PC+1+k	None	1
	CALL k	Direct Subroutine call	PC←k, STACK←PC+1	None	3

Control Signals	IF	Instructions										
		ADD	ORI	BREQ		LD		ST		CALL		
		EX1	EX1	EX1		EX1	EX2	EX1	EX2	EX1	EX2	EX3
				Z=0	Z=1							
MJ	00	xx	xx	01	01	xx	xx	xx	xx	xx	xx	10
MK	0	x	x	x	x	x	x	x	x	1	x	x
ML	0	x	x	x	x	x	x	x	x	0	x	x
IR_en	1	x	x	x	x	0	x	0	x	0	0	x
PC_en	1	0	0	0	1	0	0	0	0	x	x	1
PCh_en	0	0	0	0	0	0	0	0	0	0	0	0
PCl_en	0	0	0	0	0	0	0	0	0	0	0	0
NPC_en	1	x	x	x	x	x	x	x	x	1	0	x
SP_en	0	0	0	0	0	0	0	0	0	0	1	1
DEMUX	x	x	x	x	x	x	x	x	x	x	x	x
MA	x	1	0	x	x	x	x	x	x	x	x	x
MB	x	0	0	x	x	x	1	x	x	x	x	x
ALU_f	xxxx	0000	1001	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx	xxxx
MC	xx	00	00	xx	xx	xx	00	xx	xx	xx	xx	xx
RF_wA	0	0	0	0	0	0	0	0	0	0	0	0
RF_wB	0	1	1	0	0	0	1	0	0	0	0	0
MD	x	x	x	x	x	x	x	x	1	x	0	0
ME	x	x	x	x	x	x	1	x	1	x	0	0
DM_r	x	x	x	x	x	x	1	x	0	x	0	0
DM_w	0	0	0	0	0	0	0	0	1	0	1	1
MF	x	x	x	0	0	x	x	x	x	x	x	x
MG	xx	xx	xx	01	01	10	xx	10	xx	xx	00	00
Adder_f	xx	xx	xx	00	00	11	xx	11	xx	xx	10	10
MH	x	x	x	x	x	1	x	1	x	x	x	x
MI	x	x	x	x	x	x	x	x	x	x	0	1





Two-operand format

- - - -	- - r d	d d d d	r r r r

(e.g., ADD, CP, MOV, etc.)

Immediate format

- - - -	K K K K	d d d d	K K K K

(e.g., LDI, ANDI, ORI, etc.)

I/O format

- - - -	- A A d	d d d d	A A A A

(e.g., IN, OUT)

One-operand format

- - - -	- - - d	d d d d	- - - -

(e.g., INC, DEC, COMP, etc.)

Displacement format

- - q -	q q - d	d d d d	- q q q

(e.g., LDD, STD)

PC-relative format

- - - -	k k k k	k k k k	k k k k	k12
- - - -	- - k k	k k k k	k - - -	k7

(e.g., RJMP, RCALL, BRcc, etc.)

Direct format

- - - -	- - - k	k k k k	- - - k
k k k k	k k k k	k k k k	k k k k

(e.g., JMP, CALL)

[25 pts]

1- Consider the implementation of the SUBI Rd,K (*Subtract Immediate*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement SUBI Rd,K.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the SUBI Rd,K instruction.
  Note that this instruction takes one execute cycle (EX). Control signals for the Fetch cycle are given below. Clearly explain your reasoning.

Solution:

(a)

Only one execute cycle is needed. Similar to CPI discussed in class, subtraction is performed, however the destination register stores the result.

Stage	Micro-operations
EX	Rd ← Rd – K

(b)

Control Signals	IF	SUBI
		EX
MJ	00	xx
MK	0	x
ML	0	x
IR_en	1	x
PC_en	1	0
PCh_en	0	0
PCl_en	0	0
NPC_en	1	x
SP_en	0	0
DEMUX	x	x
MA	x	0
MB	x	0
ALU_f	xxxx	0010
MC	xx	00
RF_wA	0	0
RF_wB	0	1
MD	x	x
ME	x	x
DM_r	x	x
DM_w	0	0
MF	x	x
MG	xx	xx
Adder_f	xx	xx
MH	x	x
MI	x	x

RAL Output	SUBI
	EX
wA	x
wB	Rd
rA	Rd
rB	x

Explanation

This instruction requires an arithmetic operation and uses the datapath components outlined in Slide 18 of Lecture 4 (4.AVR_Microarchitecture.pdf) from the class notes.

EX:
Rd is read from the Register File by providing Rd to the rA address input. The immediate value K is selected as the second operand to the ALU (input B) by setting MA to 0. Note that this is the only way we can provide an immediate value as an operand to the ALU.

Subtraction is performed (ALU_f = 0010) and the result is written back to Rd using four important signals: MB is set to 0 to select the result, MC is set to 00 to select the output of MUXB, RF_wB is set to 1 load a register with the data at inB, and the wB address input is set to Rd to select the proper register to be written to.

Since we are not writing to any other registers or Data memory, RF_wA and DM_w are set to 0. All other control signals can be "don't cares" except for PC_en/PCh_en/PCl_en, and SP_en to prevent the PC register and SP register from being overwritten. Note that IR_en can be "don't care" since this is the last execute cycle and the IR register will be overwritten in the Fetch (i.e., next) cycle.

2- Consider the implementation of the ST X+, Rr (*Store Indirect and Post-Increment*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement ST X+, Rr.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the ST X+, Rr instruction.
  Note that this instruction takes two execute cycles (EX1 and EX2). Control signals for the Fetch cycle are given below. Clearly explain your reasoning.

Solution:

(a)

As with all load and store operations we require 2 cycles. This instruction performs a store with post-increment for which the first cycle uses the data path as outlined in Slide 27 of Lecture 4. Note that as we set the Data Memory pointer through DMAR, we are also incrementing the X pointer and writing the modified value back to the register file. In the second cycle we transfer the contents of the source register Rr to the location pointed to by X, which was already set in the DMAR register on the first cycle.

Stage	Micro-operations
EX1	DMAR ← Xl:Xl, Xh:Xl ← Xh:Xl + 1
EX2	M[DMAR] ← Rr

(b)

Control Signals	IF	ST X+, Rr	
		EX1	EX2
MJ	00	xx	xx
MK	0	x	x
ML	0	x	x
IR_en	1	0	x
PC_en	1	0	0
PCh_en	0	0	0
PCl_en	0	0	0
NPC_en	1	x	x
SP_en	0	0	0
DEMUX	x	x	x
MA	x	x	x
MB	x	x	x
ALU_f	xxxx	xxxx	xxxx
MC	xx	01	xx
RF_wA	0	1	0
RF_wB	0	1	0
MD	x	x	1
ME	x	x	1
DM_r	x	x	0
DM_w	0	0	1
MF	x	x	x
MG	xx	10	xx
Adder_f	xx	01	xx
MH	x	0	x
MI	x	x	x

RAL Output	ST X+, Rr	
	EX1	EX2
wA	Xh	x
wB	Xl	x
rA	Xh	x
rB	Xl	Rd

Explanation

EX1:
The contents of Xh and Xl are read from the Register file by providing Xh and Xl to rA and rB respectively. Xh:Xl (or X) is routed and latched to DMAR by setting MH to 0. Note that there is no specific signal needed to load DMAR as it is loaded every clock cycle. X is simultaneously chosen for input A of the Address Adder by setting MG to 10 to be incremented (Adder_f = 01). X+1 is written back to the register file by setting MC to 01 to select the lower half for inB. The upper half is directly connected to inA. Both RF_wA and RF_wB must be 1 to write the high and low bytes of the X register, respectively. Also, Xh and Xl are provided to wA and wB, respectively. All other control signals can be don't cares except IR_en, DM_w, PC_en, and SP_en, which need to be set to 0 to prevent IR register, Data Memory, PC register, and SP register, respectively, from being overwritten.

EX2:
Now that DMAR has been set, it is provided as the address to Data Memory through MUXE by setting ME to 1. Also the content of Rr is read from the register file by providing Rr to rB (note that Rr is the same as Rd for stores). Then Rr is written to Data Memory by setting DM_w to 1. All other control signals can be don't cares except PC_en/PCh_en/PCl_en and SP_en, which need to be set to 0 to prevent the PC register and SP register, respectively, from being overwritten. Note that IR_en can be "don't care" since this is the last execute cycle and the IR register will be overwritten in the fetch (i.e., next) cycle.

3- Consider the implementation of the RCALL (*Relative Call to Subroutine*) instruction on the enhanced AVR datapath shown below.
  (a) List and explain the sequence of microoperations required to implement RCALL.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the RCALL instruction
  Note that this instruction takes three execute cycles (EX1, EX2 and EX3). Control signals for the Fetch cycle are given below. Clearly explain your reasoning.

Solution:

(a)

RCALL involves overwriting the PC with a target address computed relative to PC+1 (the address of the next instruction). Furthermore, the return address must be stored onto the stack.

Stage	Micro-operations
EX1	M[SP] ← RARl, SP ← SP – 1
EX2	M[SP] ← RARh, SP ← SP – 1
EX3	PC ← NPC + se k

(b)

Control Signals	IF	RCALL		
		EX1	EX2	EX3
MJ	00	xx	xx	01
MK	0	x	x	x
ML	0	x	x	x
IR_en	1	0	0	x
PC_en	1	x	x	1
PCh_en	0	0	0	0
PCl_en	0	0	0	0
NPC_en	1	0	0	x
SP_en	0	1	1	0
DEMUX	x	x	x	x
MA	x	x	x	x
MB	x	x	x	x
ALU_f	xxxx	xxxx	xxxx	xxxx
MC	xx	xx	xx	xx
RF_wA	0	0	0	0
RF_wB	0	0	0	0
MD	x	0	0	x
ME	x	0	0	x
DM_r	x	0	0	x
DM_w	0	1	1	0
MF	x	x	x	0
MG	xx	00	00	01
Adder_f	xx	10	10	00
MH	x	x	x	x
MI	x	0	1	x

RAL Output	RCALL		
	EX1	EX2	EX3
wA	x	x	x
wB	x	x	x
rA	x	x	x
rB	x	x	x

1- Consider the implementation of the CPI Rd,K (*Compare Register with Immediate*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement CPI Rd,K.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the CPI Rd,K instruction.
  Note that this instruction takes one execute cycle (EX). Control signals for the Fetch cycle are given below. Clearly explain your reasoning

(a)
EX: Rd - K

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC + 1, NPC ← PC + 1, RAR ← PC + 1

(b)

Control Signals	IF	CPI
		EX
MJ	00	xx
MK	0	x
ML	0	x
IR_en	1	x
PC_en	1	0
PCh_en	0	0
PCl_en	0	0
NPC_en	1	x
SP_en	0	0
DEMUX	x	x
MA	x	0
MB	x	x
ALU_f	xxxx	0010
MC	xx	xx
RF_wA	0	0
RF_wB	0	0
MD	x	x
ME	x	x
DM_r	x	x
DM_w	0	0
MF	x	x
MG	xx	xx
Adder_f	xx	xx
MH	x	x
MI	x	x

RAL Output	CPI
	EX
wA	x
wB	x
rA	rd
rB	x

EX1: Contents of Rd is read from the register file by providing Rd to rA. The immediate value K is routed to input-B of the ALU by setting MA to 0. Then subtraction is performed but nothing is written back to the register file. The condition flags, N,Z,V,C ect will be set based on the outcome of the subtraction operation. All other control signals can be don't cares except DM_w, SP_en, and PC_en which need to be 0 to prevent overwrite. IR_en can be don't care since it's the last execute cycle

[25 pts]

4- Consider the multi-cycle implementation of the RETI (*Return from Interrupt*) instruction on the enhanced AVR datapath.
  (a) List the sequence of microoperations required to implement RETI.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the RETI instruction. Note that this instruction takes three execute cycles (EX1, EX2, and EX3) and must modify the status register (I flag), however you may ignore any required modifications to the status register for this problem.

Control Signals	IF	RETI		
		EX1	EX2	EX3
MJ	00	xx	xx	xx
MK	0	x	x	x
ML	0	x	x	x
IR_en	1	0	0	x
PC_en	1	0	0	0
PCh_en	0	0	1	0
PCl_en	0	0	0	1
NPC_en	1	x	x	x
SP_en	0	1	1	0
DEMUX	x	x	1	0
MA	x	x	x	x
MB	x	x	x	x
ALU_f	xxxx	xxxx	xxxx	xxxx
MC	x	x	x	x
RF_wA	0	0	0	0
RF_wB	0	0	0	0
MD	x	x	x	x
ME	x	x	0	0
DM_r	x	x	1	1
DM_w	0	0	0	0
MF	x	x	x	x
MG	00	00	00	xx
Adder_f	xx	01	01	xx
MH	x	x	x	x
MI	x	x	x	x

Stage	Micro-operations
EX1	SP ← SP + 1
EX2	PCh ← M[SP], SP ← SP + 1
EX3	PCl ← M[SP]

RAL Output	RETI		
	EX1	EX2	EX3
wA	x	x	x
wB	x	x	x
rA	x	x	x
rB	x	x	x

1- Consider the implementation of the RJMP (*Relative jump*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement RJMP.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the RJMP instruction.
  Note that this instruction takes one execute cycle (EX1). The Fetch cycle is shown below.

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC + 1, NPC ← PC + 1, RAR ← PC + 1

(a)
EX1: PC ← NPC + se k

(b)

Control Signals	IF	RJMP
		EX1
MJ	00	01
MK	0	x
ML	0	x
IR_en	1	x
PC_en	1	1
PCh_en	0	0
PCl_en	0	0
NPC_en	1	x
SP_en	0	0
DEMUX	x	x
MA	x	x
MB	x	x
ALU_f	xxxx	xxxx
MC	xx	xx
RF_wA	0	0
RF_wB	0	0
MD	x	x
ME	x	x
DM_r	x	x
DM_w	0	0
MF	x	0
MG	xx	01
Adder_f	xx	00
MH	x	x
MI	x	x

RAL Output	RJMP
	EX1
wA	x
wB	x
rA	x
rB	x

This instruction is nearly identical to condition Branch instruction, except the PC is always updated with the target address. PC+1 in NPC is added with the 7-bit sign-extended PC-relative displacement by setting MUXF to 0, MUXG to 01, and Adder_f to 00. The target address is then latched onto the PC by setting PC to 1 (and setting both PCh and PCl to 0's). All other control signals can be don't cares except SP_en, RF_wA and RF_wb, which are 0 to prevent overwrite. IR_en can be don't care

2- Consider the implementation of the LD Rd,Y+ (*Load Indirect and Post-Increment*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement LD Rd,Y+.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the LD Rd,Y+. instruction.
  Note that this instruction takes two execute cycles (EX1 and EX2). The Fetch cycle is shown below.

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC + 1, NPC ← PC + 1, RAR ← PC + 1

(a)
EX1: DMAR ← Yh:YL, Yh:Yl ← Yh:Yl + 1
EX2: Rd ← M[DMAR]

(b)

Control Signals	IF	LD Rd, Y+	
		EX1	EX2
MJ	00	xx	xx
MK	0	x	x
ML	0	x	x
IR_en	1	0	x
PC_en	1	0	0
PCh_en	0	0	0
PCl_en	0	0	0
NPC_en	1	x	x
SP_en	0	0	0
DEMUX	x	x	x
MA	x	x	x
MB	x	x	1
ALU_f	xxxx	xxxx	xxxx
MC	xx	01	01
RF_wA	0	1	0
RF_wB	0	1	1
MD	x	x	x
ME	x	x	1
DM_r	x	x	1
DM_w	0	0	0
MF	x	x	x
MG	xx	10	xx
Adder_f	xx	01	xx
MH	x	0	x
MI	x	x	x

RAL Output	LD Rd, Y+	
	EX1	EX2
wA	Yh	x
wB	Yl	Rd
rA	Yh	x
rB	Yl	x

EX1: The contents of Yh and Yl are read from the Register File by providing Yh and Yl to rA and rB, respectively. Yh:Yl or Y is routed to DMAR by setting MH to 0. At the same time, Y is incremented by one by the address adder by setting ALU_F to 0100 via MUXG and then latched onto YH and Yl via MUXC by setting RF_wA and RF_wB to 1s and providing Yh and Yl to wA and wB, respectively. Don't cares except DM_w, which is 0, and IR_en, PC_en, and NPC_end to 0 to prevent overwrite.

EX2: DMA is routed through MUXE and used to fetch the operand from data memory. The fetch operand is routed through MUXB and MUXC to the inB of the register file and written by setting RF_wB to 1. All others are don't cares except PC_en and SP_en, which are 0 to prevent overwrite

3- Consider the multi-cycle implementation of the RET (Return from subroutine) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement RET.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the RET instruction.
Note that this instruction takes three execute cycles (EX1, EX2, and EX3). The Fetch cycle is shown below.

Stage	Micro-operations
IF	IR ← M[PC], PC ← PC + 1, NPC ← PC + 1, RAR ← PC + 1

RAL Output	RET		
	EX1	EX2	EX3
wA	x	x	x
wB	x	x	x
rA	x	x	x
rB	x	x	x

(a)
EX1: SP ←SP +1
EX2: PCh ← M[SP], SP ← SP +1
EX3: PCl ← M[SP]

(b)

Control Signals	IF	RET		
		EX1	EX2	EX3
MJ	00	xx	xx	xx
MK	0	x	x	x
ML	0	x	x	x
IR_en	1	0	0	x
PC_en	1	0	0	0
PCh_en	0	0	1	0
PCl_en	0	0	0	1
NPC_en	1	x	x	x
SP_en	0	1	1	0
DEMUX	x	x	1	0
MA	x	x	x	x
MB	x	x	x	x
ALU_f	xxxx	xxxx	xxxx	xxxx
MC	x	x	x	x
RF_wA	0	0	0	0
RF_wB	0	0	0	0
MD	x	x	x	x
ME	x	x	0	0
DM_r	x	x	1	1
DM_w	0	0	0	0
MF	x	x	x	x
MG	00	00	00	xx
Adder_f	xx	01	01	xx
MH	x	x	x	x
MI	x	x	x	x

EX1: Content of SP is routed to input-A of the address adder by setting MG to 00, and incremented by setting Adder_f to 01. Incremented PC is then latched onto SP by setting SP_en to 1. Don't cares, except DM_w, IR_en, and PC_en, set to 0 to prevent overwrite
EX2: Content of SP is routed to input-A of the address Adder by setting MG to 00, and incremented by setting Adder_f to 01. Incremented PC is then latched onto SP by setting SP_en to 1. Data memory location pointed to by SP, ie, M[SP], which is higher byte of the return address, is read by providing SP as an address to data memory by setting ME to 0 and DM_r to 1. Read value is then routed to DEMUX to the upper byte of PC, ie, PCh by setting MUMUX to 1 and PCH_en to 1. Don't cares except DM_w, IR_en and PC_en, to prevent overwrites
EX3: Data memory located pointed to by SP, ie, M[SP] which is the lower byte of the return address, is read by providing SP as an address to the data memory by setting ME to 0 and DM_r to 1. Read value is then routed to DEMUX to the lower byte of PC, ie, PCl b setting DUMEX to 1 and PCl_en to 1. Don't cares, except DM_w and PC_en, which are 0's to prevent overwrite.

4- Consider the implementation of the PUSH (*Push Register on Stack*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement PUSH.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the PUSH instruction

Control Signals	IF	PUSH	
		EX1	EX2
MK	0	x	x
IR_en	1	0	0
PC_en	1	0	0
NPC_en	1	0	0
MG	00	x	x
MA	x	0	x
MB	xx	xx	xx
ALU_f	xxxx	0101	xxxx
MC	x	x	x
MD	xx	01	xx
RF_wA	0	1	x
RF_wB	0	1	0
ME	x	x	1
MF	x	x	0
DM_r	x	x	0
DM_w	0	0	1
Adder_f	xx	xx	xx
MH	x	x	x
MI	x	0	x
MJ	00	xx	xx
DEMUX	x	x	x
PCh_en	0	0	0
PCl_en	0	0	0

EX1: DMAR ← SP, SP ← SP -1
EX2: M[DMAR] ← Rr

RAL Output	PUSH	
	EX1	EX2
wA	SPh	x
wB	SPl	x
rA	SPh	x
rB	SPl	Rr

[35 pts]
3- Consider the implementation of the LPM (*Load Program Memory*) instruction on the enhanced AVR datapath.
(a) List and explain the sequence of microoperations required to implement LPM.
(b) List and explain the control signals and the Register Address Logic (RAL) output for the LPM instruction.
Note that this instruction takes three execute cycles (EX1, EX2, and EX3). Control signals for the Fetch cycle are given below. Clearly explain your reasoning.

(a)
EX1: PMAR ← Zh:Zl
EX2: MDR ← M[PMAR]
EX3: R0 ← MDR

RAL Output	LPM		
	EX1	EX2	EX3
wA	x	x	x
wB	x	x	R0
rA	Zh	x	x
rB	Zl	x	x

(b)

Control Signals	IF	LPM		
		EX1	EX2	EX3
MJ	00	xx	xx	xx
MK	0	x	x	x
ML	0	x	1	x
IR_en	1	0	0	x
PC_en	1	0	0	0
PCh_en	0	0	0	0
PCl_en	0	0	0	0
NPC_en	1	x	x	x
SP_en	0	0	0	0
DEMUX	x	x	x	x
MA	x	x	x	x
MB	x	x	x	x
ALU_f	xxxx	xxxx	xxxx	xxx
MC	xx	xx	xx	10
RF_wA	0	0	0	0
RF_wB	0	0	0	1
MD	x	x	x	x
ME	x	x	x	x
DM_r	x	x	x	x
DM_w	0	0	0	0
MF	x	x	x	x
MG	xx	xx or 10	xx	xx
Adder_f	xx	xx or 11	xx	xx
MH	x	0 or 1	x	x
MI	x	x	x	x

EX1: rA gets Zh and rB gets Zl to read from register file. This gets latched onto the PMAR register, by setting MH to 0. IR_en, DM_w, PC_en, and SP_en is set to 0 to prevent overwrites. Everything else should be I don't care.

EX2: Program memory gets read to PMAR by setting ML to 1, which is then sent to MDR. IR_en, DM_w, PC_en, and SP_en is set to 0 to prevent overwrites. Everything else is I don't care.

EX3: R0 gets MDR by setting MC to 10 and RF_wB to 1. DM_w, PC_en, and SP_en should be set to 0 to prevent overwrites. Everything else should be I don't care.

2- Consider the implementation of the ICALL (Indirect Call to Subroutine) instruction on the enhanced AVR datapath shown below. ICALL is similar to the RCALL (Relative Call to Subroutine) instruction, except that the Z register points to the target address.
  (a) List and explain the sequence of microoperations required to implement ICALL.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the ICALL instruction
  Note that this instruction takes two execute cycles (EX1 and EX2). Control signals for the Fetch cycle are given below. Clearly explain your reasoning.

(a) EX1: M[SP] ←RARl, SP ← SP - 1
EX2: M[SP] ←RARh, SP ← SP – 1, PC ← Z

(b)

Control Signals	IF	ICALL	
		EX1	EX2
MJ	00	xx	11
MK	0	x	x
ML	0	x	x
IR_en	1	0	x
PC_en	1	x	1
PCh_en	0	0	0
PCl_en	0	0	0
NPC_en	1	0	x
SP_en	0	1	1
DEMUX	x	x	x
MA	x	x	x
MB	x	x	x
ALU_f	xxxx	xxxx	xxxx
MC	xx	xx	xx
RF_wA	0	0	0
RF_wB	0	0	0
MD	x	0	0
ME	x	0	0
DM_r	x	0	0
DM_w	0	1	1
MF	x	x	x
MG	xx	00	00
Adder_f	xx	10	10
MH	x	x	0
MI	x	0	1

RAL Output	ICALL	
	EX1	EX2
wA	x	x
wB	x	x
rA	x	Zh
rB	x	Zl

EX1: SP needs to provide the address for data memory, so ME is set to 0. MI is set to 0 because of RAR1. To be written, MD is set to 0 and DM_w is set to 1. Adder_f is set to 10 for count, and SP_en is set to 1. IR_en and SP_en is set to 0 for overwrites. Everything else is I don't care.

EX2: SP needs to provide the address for data memory, so ME is set to 0. Since RARh is selected instead, MI must be 1 but MD and DM_w remains the same because it still gets written. Again decrement count by setting Adder_f to 10 and SP_en to 1. Zh and Zl are read in form the register file, which is sent to PC by setting MH to 0, MJ set to 11, and PC_en to 1. Everything else should be I don't care.

[30 pts]
1- Consider the implementation of the LD Rd, -X (*Load Indirect and Pre-decrement*) instruction on the enhanced AVR datapath.
  (a) List and explain the sequence of microoperations required to implement LD Rd, -X.
  (b) List and explain the control signals and the Register Address Logic (RAL) output for the LD Rd, -X instruction. Some of the control signals are given below.
  Note that this instruction takes two execute cycles (EX1 and EX2). Control signals for the Fetch cycle are given below. Clearly explain your reasoning.

(a)
EX1: DMAR ← Xh:Xl - 1, Xh:Xl ← Xh:Xl - 1
EX2: Rd ← M[DMAR]

(b)

Control Signals	IF	LD Rd, -X	
		EX1	EX2
MJ	00	xx	xx
MK	0	x	x
ML	0	x	x
IR_en	1	0	x
PC_en	1	0	0
PCh_en	0	0	0
PCl_en	0	0	0
NPC_en	1	x	x
SP_en	0	0	0
DEMUX	x	x	x
MA	x	x	x
MB	x	x	1
ALU_f	xxxx	xxxx	xxxx
MC	xx	01	00
RF_wA	0	1	0
RF_wB	0	1	1
MD	x	x	x
ME	x	x	1
DM_r	x	x	1
DM_w	0	0	0
MF	x	x	x
MG	xx	10	xx
Adder_f	xx	10	xx
MH	x	1	x
MI	x	x	x

RAL Output	LD Rd, -X	
	EX1	EX2
wA	Xh	x
wB	Xl	Rd
rA	Xh	x
rB	Xl	x

EX1: Contents of Xh and Xl are read from the register file by providing Xh and Xl to rA and rB, respectively. Xh:Xl is dcremented by one by the address adder by setting Adder_f to 10 via MUXG and then latched onto Xh and Xl via MUXC by setting both RF_wA and RF_wB to 1s and providing Xh and Xl to wA and wB, respectively. X is routed to DMAR by setting HM to 1. Don't cares except DM_w to 0 and IR_en, PC_en, and NPC_en to 0 to prevent overwrite.

EX2: DMAR is routed through MUXE and used to fetch the operand from data memory. Fetch operand is routed through MUXB and MUXC to the inB of the register file and written by setting RF_wB to 1. Don't cares except PC_en and SP_en to 0 to prevent overwrite.

Consider the implementation of the `LDD Rd, Y+q` (*Load Indirect with Displacement Using Index Y*) instruction on the enhanced AVR datapath shown on the following page.
(a) List and explain the sequence of microoperations required to implement `LDD`.
(b) List and explain the control signals and the Register Address Logic (RAL) output for the `LDD` instruction. Some of the control signals are given below.
Note that this instruction takes two execute cycles (EX1 and EX2). The Fetch cycle is shown below.

IF: IR ← M[PC], PC ← PC +1, NPC ← PC +1, RAR ← PC + 1

(a)
EX1: DMAR ← Y+q   Setting the pointer, i.e. placing the address computed by Y+q into the DMAR
EX2: Rd ← M[DMAR]   Loading the value pointed to by Y+q into the destination register

(b)

Control Signals	IF	LDD	
		EX1	EX2
MK	0	x	x
IR_en	1	0	0
PC_en	1	0	0
NPC_en	1	0	0
MG	00	xx	xx
MA	x	0	x
MB	xx	00	xx
ALU_f	xxxx	0000	xxxx
MC	x	x	1
MD	xx	xx	0
RF_wA	0	0	0
RF_wB	0	0	1
ME	x	x	x
MF	x	x	0
DM_r	x	x	1
DM_w	0	0	0
Adder_f	xx	xx	xx
MH	x	x	x
MI	x	1	x
MJ	00	xx	xx
DEMUX	x	x	x
PCh_en	0	0	0
PCl_en	0	0	0

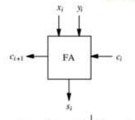
RAL Output	LDD	
	EX1	EX2
wA	x	x
wB	x	Rd
rA	Yh	x
rB	Yl	x

Explanation of control signals

EX1: To add the q offset to address register Y, we must provide both values to the 16-bit ALU given in the datapath. The only way Y can be used as an operand to the ALU is through MUXA coming from the concatenation unit. First the high (Yh) and low (Yl) addresses of the Y register are generated by the RAL for rA and rB respectively. Then Y is chosen by setting MA=0. We also notice q must pass through MUXB via input 0, by setting MB=00. The addition operation is already set for us (ALU_f=0000). The output of the ALU (Y+q) must be routed to the DMAR through MUXI by setting MI=1. To prevent changes to registers and memory, RF_wA, RF_wB, and DM_w must all be set to zero. All other control signals to be filled out can be don't cares.

EX2: Since we are loading a value from Data Memory, we select the appropriate address coming from DMAR through MUXF by setting MF=0. Also, to read from Data Memory, DM_r=1 and DM_w=0. To route the 8-bit value that is read from Data Memory to the appropriate register Rd, only one path exists leading to inB of the register file. So RAL generates the address for Rd on wB. The two multiplexers MUXC and MUXD then route the value to be loaded into the destination register by setting MC=1 and MD=0. To allow for the input on inB to be written to the destination register, RF_wB=1. Since we do not modify any other registers, RF_wB=0. All other control signals can be don't cares.

Full Adder



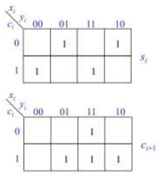
$$s_i = x_i \oplus y_i \oplus c_i$$

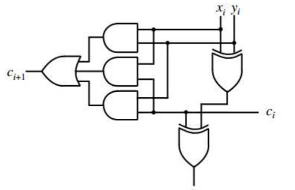
$$c_{i+1} = x_iy_i + (x_i \oplus y_i)c_i$$

or

$$c_{i+1} = x_iy_i + (x_i + y_i)c_i$$

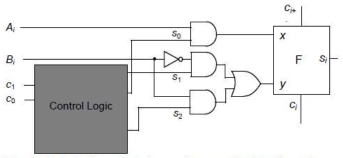
x_i	y_i	c_i	c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1





3- Consider i^{th} bit-slice of a n-bit arithmetic unit with A_i and B_i as inputs and three controls signals c_1, c_0, and c_{in}, where c_{in} is the carry-in to the n-bit arithmetic unit (not shown). *Design* the control logic required implement the following set of arithmetic operations. That is, define the truth table, K-map, and realization for the control logic to implement n-bit arithmetic unit. *Your design must result in minimum number of logic gates.*

c_1	c_0	c_{in}	Operation
0	0	0	F = A + B
0	0	1	F = A + B + 1
0	1	0	F = A
0	1	1	F = A + 1
1	0	0	F = B'
1	0	1	F = B'+1
1	1	0	F = A + B'
1	1	1	F = A + B' +1



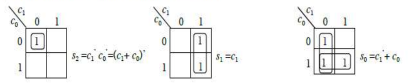
We can define the truth table by looking at the requirement for s_2, s_1, and s_0 depending on the operation. Thus, we have the following truth table

Input				Output		
c_1	c_0	c_{in}	Operation	s_2	s_1	s_0
0	0	0	F = A + B	1	0	1
0	0	1	F = A + B + 1	1	0	1
0	1	0	F = A	0	0	1
0	1	1	F = A + 1	0	0	1
1	0	0	F = B'	0	1	0
1	0	1	F = B'+1	0	1	0
1	1	0	F = A + B'	0	1	1
1	1	1	F = A + B' +1	0	1	1

Note that control signals s_2, s_1, and s_0 are the same regardless whether s_{in} is 0 or 1. Therefore, the truth table can be simplified to

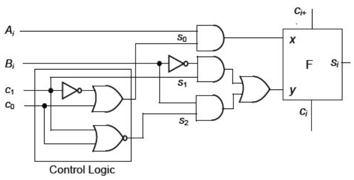
Input		Output		
c_1	c_0	s_2	s_1	s_0
0	0	1	0	1
0	1	0	0	1
1	0	0	1	0
1	1	0	1	1

This is can be minimized using the following three K-maps



$s_2 = c_1' c_0' = (c_1 + c_0)'$ $s_1 = c_1$ $s_0 = c_1' + c_0$

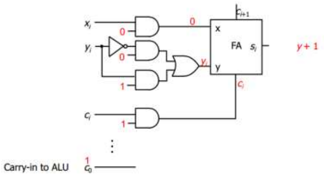
Thus, we have the following implementation for each stage or bit slice.



Arithmetic - Example 1

ARITHMETIC

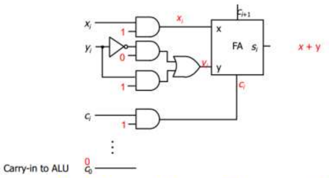
M	$S_2S_1S_0$	C_0	Micro Operation	Description
1	0 0 1	1	s = y + 1	Increment y



Arithmetic - Example 4

ARITHMETIC

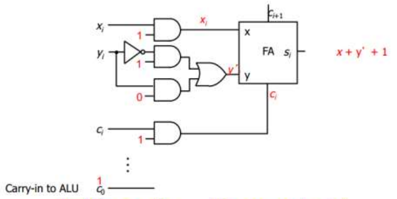
M	$S_2S_1S_0$	C_0	Micro Operation	Description
1	1 0 1	0	s = x + y	Add x and y



Arithmetic - Example 6

ARITHMETIC

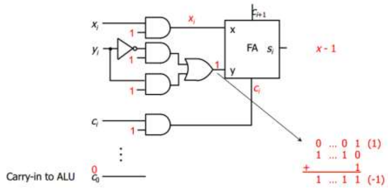
M	$S_2S_1S_0$	C_0	Micro Operation	Description
1	1 1 0	1	s = x + y' + 1	x plus 2' s complement of y



Arithmetic - Example 7

ARITHMETIC

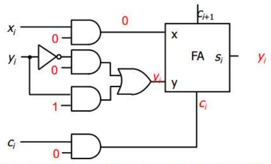
M	$S_2S_1S_0$	C_0	Micro Operation	Description
1	1 1 1	0	s = x - 1	Decrement x



Logic - Example 2

ARITHMETIC

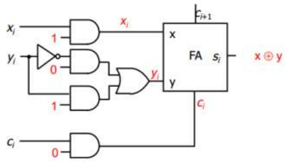
M	$S_2S_1S_0$	Micro Operation	Description
0	0 0 1	s = y	Transfer y



Logic - Example 6

ARITHMETIC

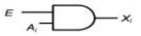
M	$S_2S_1S_0$	Micro Operation	Description
0	1 0 1	s = x ⊕ y	EOR



Design a 4-bit arithmetic circuit with three control signals S_1, S_0, and C_{in} (carry-in) using a 4-bit ripple-car adder and logic gates, which performs the following arithmetic operations:

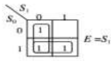
S_1	S_0	C_{in}	Operation
0	0	0	F = A + B
0	0	1	F = A + B + 1
0	1	0	F = A
0	1	1	F = A + 1
1	0	0	F = B'
1	0	1	F = B'+1
1	1	0	F = A + B'
1	1	1	F = A + B' +1

based on the design we saw in class, we can define the truth table by looking at the requirement for X_i and inputs to each FA. For X_i, we need an AND gate with A_i as one input and some control signal, say E, th control whether X_i=A_i. This is shown below.



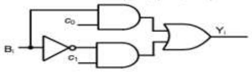
This is can be defined by the following truth table and realization:

Input		Output
S_1	S_0	E
0	0	1
0	1	1
1	0	0
1	1	1



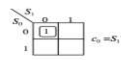
$E = S_1' + S_o$

For Y_i, we need a MUX that determines whether $Y_i = B$, $Y_i = B'$, or Y_i= 0. This is done by a logic shown below:

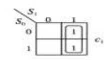


Based on this, we can define the following truth table for c_o and c_1:

Input		Output	
S_1	S_0	c_o	c_1
0	0	1	0
0	1	0	0
1	0	0	1
1	1	0	1

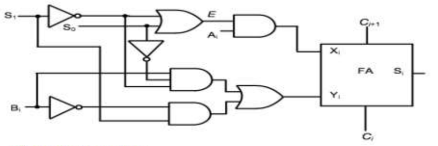


$c_0 = S_1' S_o'$



$c_1 = S_1$

Finally, we can implement "+1" by setting C_{in}=1 (i.e., carry into the adder). Thus, we have the followin implementation for each stage or bit slice.



Carry Propagation

- 2' s complement is the best.
- 1' s complement twice as long.
- Significant delay reduction using carry look ahead concept.
  - Example - 64 bit adder - reduced from 130 gate delays to 14, or improved by a factor of 8

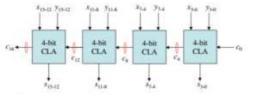
CLA Equations

$c_1 = g_0 + p_0c_o$
$c_2 = g_1 + p_1g_0 + p_1p_0c_o$
$c_3 = g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_o$
$c_4 = g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_o$

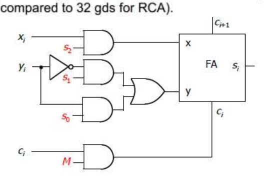
Can be implemented with two-level logic! (2gds)

- 0th Stage:
  - $s_0 = x_0 \oplus y_0 \oplus c_0$
  - $c_1 = g_0 + p_0c_o$
- 1st Stage:
  - $s_1 = x_1 \oplus y_1 \oplus c_1$
  - $c_2 = g_1 + p_1c_1 = g_1 + p_1(g_0 + p_0c_o) = g_1 + p_1g_0 + p_1p_0c_o$
- 2nd Stage
  - $s_2 = x_2 \oplus y_2 \oplus c_2$
  - $c_3 = g_2 + p_2c_2 = g_2 + p_2(g_1 + p_1g_0 + p_1p_0c_o)$
    $= g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_o$
- 3rd Stage
  - $s_3 = x_3 \oplus y_3 \oplus c_2$
  - $c_4 = g_3 + p_3c_3 = g_3 + p_3(g_2 + p_2g_1 + p_2p_1g_0 + p_2p_1p_0c_o)$
    $= g_3 + p_3g_2 + p_3p_2g_1 + p_3p_2p_1g_0 + p_3p_2p_1p_0c_o$

Cascade CLA Adder



- Total delay:
  - 1 gd for g_i' s and p_i' s for i = 0, 1, …, n-1
  - 2 gds x 3 for c_4, c_8, c_{12}
  - 2 gds for c_{13}, c_{14}, c_{15} (after c_{12} is available)
  - 2 gds for $s_{15\text{-}12}$
- For m CLA stages, total delay = 3 + 2m. For m=4, 11 gds (compared to 32 gds for RCA).



(a) Bit-slice ALU.

ARITHMETIC

M	$S_2S_1S_0$	C_0	Micro Operation	Description
1	0 0 1	1	s = y + 1	Increment y
1	0 1 0	1	s = y' + 1	2's complement y
1	1 0 0	1	s = x + 1	Increment x
1	1 0 1	0	s = x + y	Add x and y
1	1 1 0	0	s = x + y'	x plus 1's complement of y
1	1 1 0	1	s = x + y' + 1	x plus 2's complement of y
1	1 1 1	0	s = x - 1	Decrement x

(b) Arithmetic operations.

LOGIC

M	$S_2S_1S_0$	Micro Operation	Description
0	0 0 0	s = 0	Clear
0	0 0 1	s = y	Transfer y
0	0 1 0	s = y'	Comp. y
0	0 1 1	s = 1	Set
0	1 0 0	s = x	Transfer x
0	1 0 1	s = x⊕y	EOR
0	1 1 0	s = (x⊕y)'	ENOR
0	1 1 1	s = x'	Comp. x

(c) Logic operations.

2- Consider the following AVR instruction code sequence for the 16-bit multiplier from Lab 3:

```
MUL16_ILOOP:   ld     A, X+         ; Get byte of A operand
               ld     B, Y          ; Get byte of B operand
               mul    A,B           ; Multiply A and B
               ld     A, Z+         ; Get a result byte from memory
               ld     B, Z+         ; Get the next result byte from memory
               add    rlo, A        ; rlo <= rlo + A
               adc    rhi, B        ; rhi <= rhi + B + carry
               ld     A, Z          ; Get a third byte from the result
               adc    A, zero       ; Add carry to A
               st     Z, A          ; Store third byte to memory
               st     -Z, rhi       ; Store second byte to memory
               st     -Z, rlo       ; Store third byte to memory
               adiw   ZH:ZL, 1      ; Z <= Z + 1
               dec    iloop         ; Decrement counter
               brne   MUL16_ILOOP   ; Loop if iLoop != 0
                                    ; End inner for loop
```

Solution:

(a) Since all loads and stores are 2 cycles each, we only list the remaining instructions:

`mul, adiw`: 2 cycles
`add, adc, dec`: 1 cycle
`brne`: 1 cycle when condition is false, 2 cycles when condition is true (for pipelining)

`brne` is similar to `breq`; whether one or two execute cycles take place is specifically related to pipelining. As discussed on Slide 64 of the class notes, a one-cycle branch penalty is incurred if the branch is taken (i.e., condition is true) or not.

(b) Non-pipelined:
The fetch and execute cycles for each instruction are listed below. Note for the branch instruction one execute cycle is used regardless if the branch is taken or not. This is because pipelining is not employed so any given instruction cannot be fetched until the instruction preceding it has completed the execute cycle.

Instruction	Fetch	EX	Total
`ld    A, X+`	1	2	3
`ld    B, Y`	1	2	3
`mul   A,B`	1	2	3
`ld    A, Z+`	1	2	3
`ld    B, Z+`	1	2	3
`add   rlo, A`	1	1	2
`adc   rhi, B`	1	1	2
`ld    A, Z`	1	2	3
`adc   A, zero`	1	1	2
`st    Z, A`	1	2	3
`st    -Z, rhi`	1	2	3
`st    -Z, rlo`	1	2	3
`adiw  ZH:ZL, 1`	1	2	3
`dec   iloop`	1	1	2
`brne  MUL16_ILOOP`	1	1	2

The total number of cycles amounts to 40.

(c) Pipelined:
For the pipelined version, after the first fetch, each Fetch cycle is overlapped with the last execute cycle of the preceding instruction. Also, for the brne instruction, since the condition is true, we will have a one-cycle branch penalty resulting in two execute cycles.

Instruction	Fetch	EX	Total
`ld    A, X+`	1	2	3
`ld    B, Y`	-	2	2
`mul   A,B`	-	2	2
`ld    A, Z+`	-	2	2
`ld    B, Z+`	-	2	2
`add   rlo, A`	-	1	1
`adc   rhi, B`	-	1	1
`ld    A, Z`	-	2	2
`adc   A, zero`	-	1	1
`st    Z, A`	-	2	2
`st    -Z, rhi`	-	2	2
`st    -Z, rlo`		2	2
`adiw  ZH:ZL, 1`	-	2	2
`dec   iloop`	-	1	1
`brne  MUL16_ILOOP`	-	2	2

With pipelining, the total number of cycles is now 27.

(c) The performance improvement is (40-27)/40 = .325 = 32.5% .