ECE 375
Computer Organization and Assembly Language Programming
Fall 2015

Solutions Set #1

[25 pts]

1- Consider the pseudo-CPU discussed in class with a memory unit with 4K words of 16 bits each. An instruction is stored in one word of memory. The instruction format is divided into three fields: opcode field, addressing mode field that specifies direct or indirect addressing mode, and an address field.

(a) What is the maximum number of opcodes that can be incorporated into the CPU? How many bits are in the opcode field, the addressing mode field, and the address field? Draw the instruction format and indicate the number of bits in each field.

(b) How many bits are in the registers PC, MAR, MDR, IR, and AC?

Solution

(a) Since the memory contains 4K (2^{12}), 12 bits are required for the address field. One bit is needed to specify direct or indirect addressing mode. This leaves 16 minus 12 + 1 (addressing mode) = 3, or 3 bits for the opcode field. Thus, this allows $2^3 = 8$ different opcodes (or instructions). The instruction format is shown below.

3	1	12
Opcode	I	Address

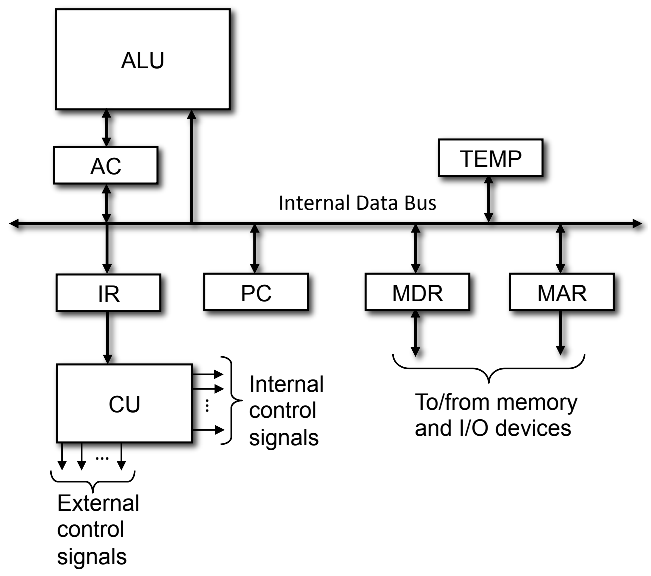
(b) From (a) we see that 12 bits are required to address the entire memory, thus MAR and PC each have 12 bits. Since a memory words consists of 16 bits MDR and AC each have 16 bits. The number of bits required for IR depends on whether you assume the IR can hold the entire instruction or just the opcode plus the addressing mode. If IR can hold the entire instruction then it is 16 bits. If it just holds the opcode and the addressing mode, then it is 4 bits.

[25 pts]

2- Consider the following hypothetical 1-address assembly instruction called "Store Accumulator Indirect with Post-increment" of the form

```
STA     (x)+      ; M(M(x)) ← AC, M(x) ← M(x)+1
```

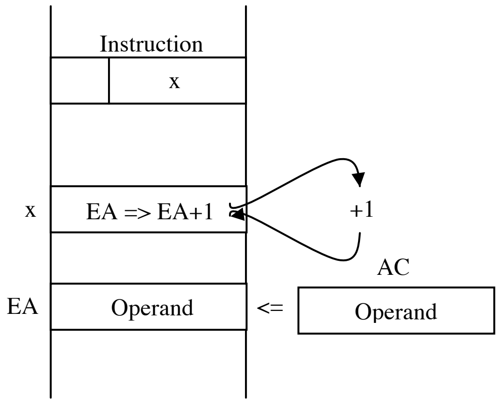
Suppose we want to implement this instruction on the pseudo-CPU discussed in class augmented with a temporary register TEMP. An instruction consists of 16 bits: A 4-bit opcode and a 12-bit address. All operands are 16 bits. PC and MAR each contain 12 bits. AC, MDR, and TEMP each contain 16 bits, and IR is 4 bits. Give the sequence of *microoperations* required to implement the Execute cycle (Fetch cycle is given below) for the above STA (x)+ instruction. Your solution should result in exactly 8 microoperations. Assume PC is currently pointing to the STA (x)+ instruction and only PC and AC have the capability to increment/decrement itself.



Fetch Cycle
Step 1: MAR ← PC;
Step 2: MDR ← M(MAR), PC ← PC+1 ; Read inst. & increment PC
Step 3: IR ← $\text{MDR}_{\text{opcode}}$, MAR← $\text{MDR}_{\text{address}}$

Solution

The graphical representation of STA (x)+ is shown below.



These steps can be implemented by the following sequence of microoperations:

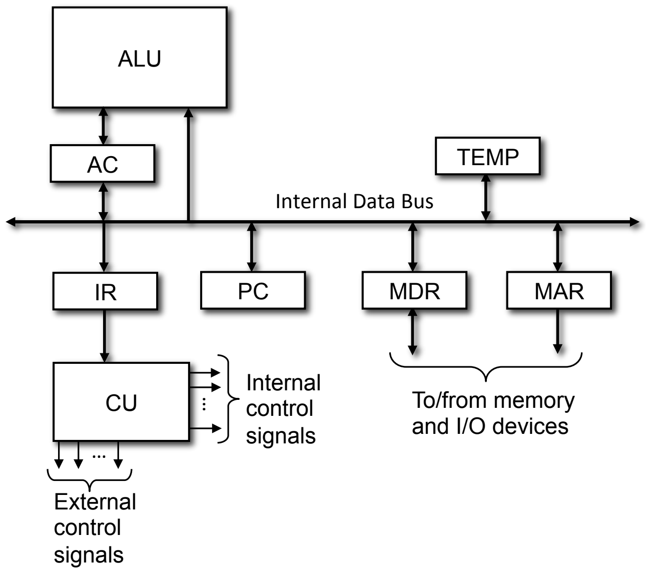
Execute Cycle
Step 1: MDR ← M(MAR), TEMP ← AC ; Get EA from memory (i.e., M(x)), Save AC to Temp
Step 2: AC ← MDR ; Increment EA
Step 3: AC ← AC + 1 ;
Step 4: MDR ← AC ; Store EA+1 into M(x)
Step 5: M(MAR) ← MDR, AC ← AC -1 ; Get back EA

Step 6: MAR ← AC ; Have MAR point to EA
Step 7: MDR ← Temp, AC ← Temp ; Restore Temp to AC, and move Temp to MDR
Step 8: M(MAR) ← MDR ; Store content of MDR, i.e., AC, into M(M(x))=M(EA)

[25 pts]
3- The PDP-8 was the first successful commercial minicomputer, produced by Digital Equipment Corporation (DEC) in the 1960s (see http://en.wikipedia.org/wiki/PDP-8). Two of the assembly instructions it supported were of the form:

(a) Deposit and clear the accumulator: DCA Y ; M(Y) ← AC, AC ← 0
(b) Jump to subroutine: JMS Y ; M(Y) ← PC, PC ← Y + 1

Suppose the pseudo-CPU discussed in class with a temporary register (TEMP) shown below can be used to implement DCA and JMS instructions. Give the sequence of *microoperations* required to fetch and execute DCA and JMS. Your solution should result in minimum number of microoperations. Assume PC is currently pointing to the instruction and only PC and AC have the capability to increment itself.



Solution

(a) and (b)

Fetch Cycle
Step 1: MAR ← PC;
Step 2: MDR ← M(MAR), PC ← PC+1 ; Read inst. & increment PC
Step 3: IR ← $\text{MDR}_{\text{opcode}}$, MAR ← $\text{MDR}_{\text{address}}$

(a)
Execute Cycle: DCA Y
Step 1: MDR ← AC ; Transfer content of AC to MDR
Step 2: M(MAR) ← MDR, AC ← 0 ; Store AC into M(Y), and at the same time clear AC

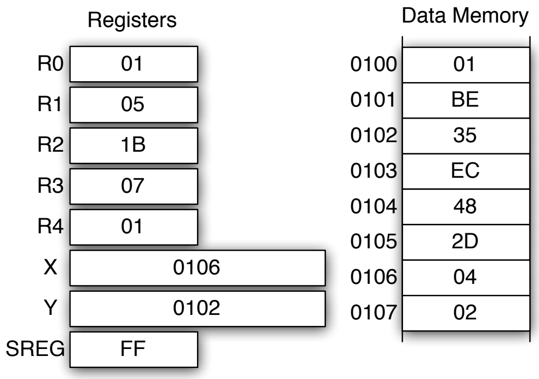
(b)
Execute Cycle: JMS Y
Step 1: MDR ← PC ; Transfer contents of PC to MDR
Step 2: M(MAR) ← MDR, PC ← MAR ; Store PC into M(Y) and at transfer content of MAR to PC

Step 3:   PC ← PC + 1   ; Increment PC

[25 pts]

4- Based on the initial register and data memory contents shown below (represented in hexadecimal), show how these contents are modified (in *hexadecimal*) after executing each of the following AVR assembly instructions. Do not be concerned about what happens to the Status Register (SREG) *after* the operation. *Instructions are unrelated.*

```
(i)   MOV      R1, R28
(ii)  LD       R4, Y+
(iii) LDI      R4, 33
(iv)  MUL      R2, R3
(v)   ROL      R3
```



Solution:

(i) R28 (which is also the lower 8 bits of Y register) contains 02_{16}. Thus, R1 changes to 02_{16}. [Extra] No flags are modified.

(ii) Effective address is 0102_{16}, thus $M[Y]=35_{16}$. R4 changes 35_{16} and Y changes to 0103_{16}. [Extra] N, Z, and C flags are not modified.

(iii) LDI instruction only allows R31-R16 as the destination. Thus, this instruction is not allowed.

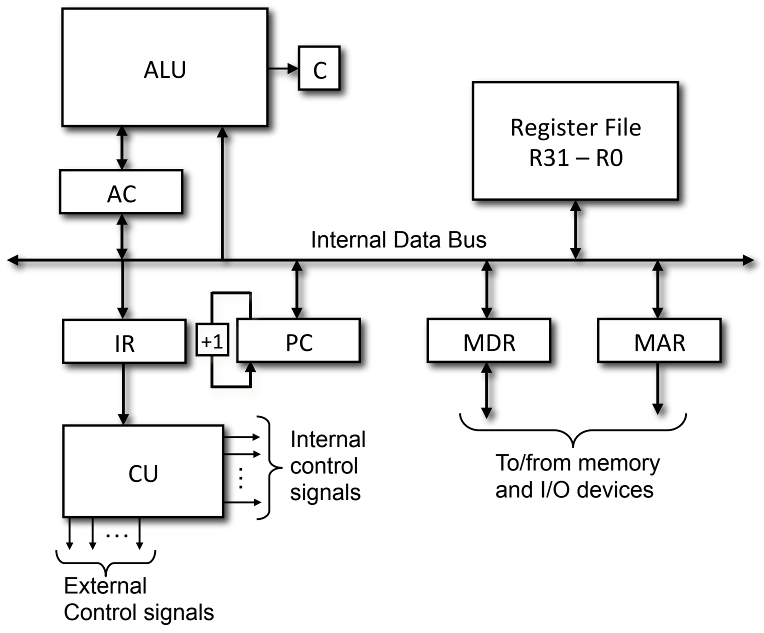
(iv) Since $1B_{16} \times 07_{16} = 27_{10} \times 07_{10} = 189_{10} = 00BD_{16}$
Since High byte of the result is stored in R1 and low byte of the result is stored in R0,
R1 changes to 00 and R0 changes to BD_{16}. [Extra] C-bit is cleared since the 15th bit is zero, and Z-bit is cleared since the result is non-zero, i.e., SR= 11111100_2=FC_{16}. N-bit is not modified.

(v) Since C-bit = 1 (0th bit), so R3 = 00001111_2 =$0F_{16}$. [Extra] N-bit is cleared since the result is positive (i.e., MSB of the result is zero), Z-bit is cleared since the result is non-zero, and C-bit is cleared since MSB of the R3 before the shift was zero. Thus, SR = 111111000_2=$F8_{16}$.

ECE 375
Computer Organization and Assembly Language Programming
Fall 2015
Solutions Set #2

[25 pts]

1- Consider the internal structure of the pseudo-CPU discussed in class augmented with a *single-port register file* (i.e., only one register value can be read at a time) 32 8-bit registers (R31-R0) and a carry bit (C-bit), which is set/reset after each arithmetic operation. Suppose the pseudo-CPU can be used to implement the AVR instruction `ADIW ZH:ZL,32` (Add immediate to word). `ADIW` is a 16-bit instruction, where the upper byte represents the opcode and the lower byte represents an immediate value, i.e., "`32`" (do not worry about the fact that the actual format is slightly different). Give the sequence of microoperations required to Fetch and Execute the `ADIW` instruction. *Your solutions should result in exactly 5 cycles for the fetch cycle* and *6 cycles for the execute cycle.* Assume the memory is organized into addressable bytes (i.e., each memory word is a byte), MDR, IR, and AC registers are 8-bit wide, and PC and MAR registers are 16-bit wide. Also, assume Internal Data Bus is 16-bit wide and thus can handle 8-bit or 16-bit (as well as portion of 8-bit or 16-bit) transfers in one microoperation and only PC and AC have the capability to increment itself.



__Solution__

Since `ADIW ZH:ZL,32` is a 16-bit instruction and memory is organized into consecutive bytes, the instruction occupies two consecutive bytes. Thus, two memory accesses are needed to fetch the instruction into the IR.

<u>Fetch cycle</u>

Step 1:	MAR ← PC;	
Step 2:	MDR ← M(MAR), PC ← PC+1	; Get the high byte of the instruction and increment PC
Step 3:	IR ← MDR	; At this point, CU knows this is ADIW
Step 4:	MAR ← PC;	
Step 5:	MDR ← M(MAR), PC ← PC+1	; Get the low byte of the instruction and increment PC

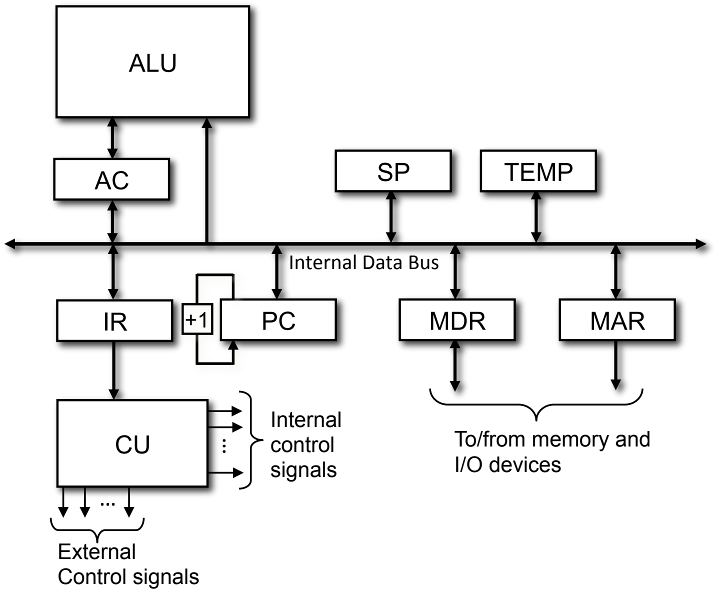
<u>Execute cycle</u>

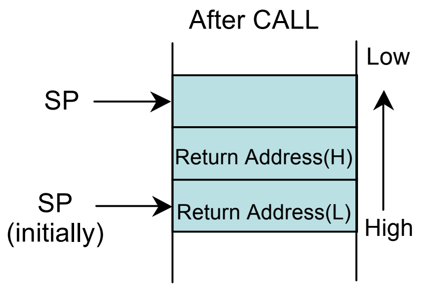
Step 6: AC ← R30

```
Step 7:  AC ← AC + MDR              ; Add 32 to ZL
Step 8:  R30 ← AC                   ; Write back to register file
Step 9:  AC ← R31
Step 10: If (C==1) then AC ← AC +1  ; Increment ZH if there was a carry
Step 11: R31 ← AC                   ; Write it back to register file
```

[25 pts]

2- Consider the internal structure of the pseudo-CPU discussed in class augmented with a Stack Pointer (SP) and a 16-bit Temporary (TEMP) registers. Suppose the pseudo-CPU can be used to implement the AVR instruction CALL *label* (Long Call to a Subroutine). CALL *label* is a four-byte instruction. Give the sequence of microoperations required to Fetch and Execute CALL *label*. *Your solution should result in no more than 12 microoperations for the fetch cycle (i.e., fetch the 16-bit opcode into the IR and the 16-bit address into TEMP) and 7 microoperations for the execute cycle).* You may assume the SP register has the capability to increment/decrement itself. Assume the PC is currently pointing to the CALL instruction and MDR register is 8-bit wide, and SP, PC, IR, and MAR are 16-bit wide. Note that since PC is 16 bits, only the lower 16 bits of the target address (i.e., *label*) are used. In other words, the upper 16 bits represent the opcode and the lower 16 bits represent the target address for the subroutine call. Also, assume Internal Data Bus is 16-bit wide and can handle 8-bit or 16-bit transfers in one microoperation. Clearly state any other assumptions made.





Solution:

Since MDR is only 8 bits, the memory is organized into addressable bytes.
; Fetch the first two bytes of the instruction

```
Step 1:  MAR ← PC;
Step 2:  MDR ← M(MAR), PC ← PC+1     ; Get the  high byte of the instruction and increment PC
Step 3:  IR(15…8) ← MDR
Step 4:  MAR ← PC;
Step 5:  MDR ← M(MAR), PC ← PC+1     ; Get the low byte of the instruction and increment PC
Step 6:  IR(7…0)  ← MDR              ; At this point, CU knows this is a CALL
; 16–bit target address is fetched
Step 7:  MAR ← PC;
Step 8:  MDR ← M(MAR), PC ← PC+1
Step 9:  TEMP(15…8) ← MDR
Step 10: MAR ← PC;
Step 11: MDR ← M(MAR), PC ← PC+1
Step 12: TEMP(7…0) ← MDR

; The return address is pushed onto the stack
Step 13: MDR ← PC(7…0)
Step 14: MAR ← SP
Step 15: M(MAR) ← MDR, SP ← SP-1     ; Push the lower byte of return address onto stack
Step 16: MDR ← PC(15…8)
Step 17: MAR ← SP
Step 18: M(MAR) ← MDR, SP ← SP-1     ; Push the higher byte of return address onto stack

: Put H and L addresses of the target address to the PC
Step 19: PC ← TEMP              ; Put the TA into PC (points to address of subroutine)

Goto fetch and Execute cycle.
```

[25 pts]

3- Consider the following code written in AVR assembly. Explain in words what the program accomplishes when it is executed. That is, explain what it does, how it does it, and how many times it does it. What is the value of location CTR when the execution completes?

```
.ORG   $0000
       RJMP   START
.ORG   $0046
START: LDI    XH, high(CTR)
       LDI    XL, low(CTR)
       LDI    R31, 0xf0
       CLR    R5
LOOP:  CLC
       ROL    R31
       BRCC   SKIP
       INC    R5
SKIP:  CPI    R31, 0x00
       BRNE   LOOP
       ST     X, R5
DONE:  JMP    DONE
.DSEG
.ORG   $0100
CTR:   .BYTE  1
```

Solution

At start up, the RJMP instruction is executed, which causes the program to start executing at location START. Then, the program loads the value $f0_{16}=11110000_2$ and rotates a bit at a time through the carry to determine the number of 1's. The number of 1's is stored at location CTR. Since 11110000_2 has 4 1's, the value at location CTR will be 4. This is done by

- Loading immediate value $f0 into R31
- Rotating R31 left through the carry to see if the shifted bit is set, and feed in 1's.
- If set, increment counter (R5).
- BRNE checks if the loop should terminate. And the program terminates when R31 becomes all 0's. The number times the loop executes depends on number 1's. For the value $f0_{16}$, the loop executes 4 times).
- BYTE assembler directive allocates 1 byte for storing the count.

[25 pts]

4- Consider the following code discussed in Problem #3. Show the binary representation of the code in program memory. That is, show the binary representation of each instruction at its location in program memory (in hexidecimal).

Solution

```
.ORG   $0000                         Address           Binary
       RJMP   START                  0000:      1100   0000   0100   0101
.ORG   $0046
START: LDI    XH, high(CTR)          0046:      1110   0000   1011   0001
       LDI    XL, low(CTR)           0047:      1110   0000   1010   0000
       LDI    R31, 0xf0              0048:      1110   1111   1111   0000
       CLR    R5                     0049:      0010   0100   0101   0101
LOOP:  CLC                           004A:      1001   0100   1000   1000
       ROL    R31                    004B:      0001   1111   1111   1111
       BRCC   SKIP                   004C:      1111   0100   0000   1000
       INC    R5                     004D:      1001   0100   0101   0011
SKIP:  CPI    R31, 0x00              004E:      0011   0000   1111   0000
       BRNE   LOOP                   004F:      1111   0111   1101   0001
       ST     X, R5                  0050:      1001   0010   0101   1100
DONE:  JMP    DONE                   0051:      1001   0100   0000   1100
                                     0052:      0000   0000   0101   0001
.DSEG
.ORG   $0100
CTR:   .BYTE  1
```

Highlights:

- Location 0000: Branch target address is 0046, thus $0046_{16} - 0001_{16} = 0045_{16} = 000001000101_2$ => kkkk kkkk kkkk = 0000 0100 0101.
- Location 0046: high(CTR) = 01_{16} thus KKKK KKKK = 0000 0001, and XH = R27 = dddd= 11011 (with implied 1 in the MSB).
- Location 0047: low(CTR) = 00_{16} thus KKKK KKKK = 0000 0000, and XL = R26 = dddd= 11010 (with implied 1 in the MSB).
- Location 0048: dddd=11111 (with implied 1 in the MSB), and KKKK KKKK = 1111 0000.
- Location 0049: d dddd=0 0101 and d dddd=0 0101 (note that first one indicates red locations and second one indicates blue locations).
- Location 004A: Straight from the AVR manual.
- Location 004B: r rrrr=1 1111, d dddd=1 1111, which is equivalent to ADC Rd, Rd.
- Location 004C: Branch target address is 004E, thus $004E_{16}$-$004D_{16}$= 0001_{16}=0000001_2 => kk kkkk k = 00 0000 1
- Location 004D: R5 = d dddd= 0 0101
- Location 004E: dddd = 11111 (with implied 1 in the MSB), and KKKK KKKK = 0000 0000
- Location 004F: Branch target address is 004A, thus $004A_{16}$-0050_{16}= -0006_{16}=1111010_2 => kk kkkk k = 11 1101 0
- Location 0050: R5 = d dddd = 0 0101
- Location 0051: Jump target address is 0051 = 0000 0000 0101 0001
  => k kkkk k kkkk kkkk kkkk kkkk = 0 0000 0 0000 0000 0101 0001

ECE 375
Computer Organization and Assembly Language Programming
Fall 2015
Solutions Set #3

[25 pts]

1- The AVR code below (with some missing information) is designed to initialize and service interrupts from three I/O devices (DevA, DevB, and DevC).
   (a) There are 8 external interrupt pins (INT7-INT0) in AVR. Which three interrupt pins are these I/O devices connected to?
   (b) Which I/O device's interrupt is detected on a falling edge?
   (c) The interrupt pins referred to in part (a) are connected to two of the 7 ports (PORTA-PORTG) in AVR. Which ports are they?
   (d) There are important instructions missing in lines 1-2 of the code. Fill in the missing instructions in lines 1-2 so that the code will work correctly. Note that DevB requires the pull-up resistor to be activated.
   (e) Suppose DevA requires that no interrupts can be latched onto the External Interrupt Flag Register (EIFR) while it is being serviced. Fill in lines 3-4 and 5-6 with the necessary code to mask out all other interrupts at the beginning of `ISR_DevA` and then reset the EIMSK so that latching of interrupts from the three devices can resume just before returning from `ISR_DevA`.
   (f) Suppose interrupts are detected from all three interrupt pins at the same time. Which subroutine (`ISR_DevA`, `ISR_DevB`, or `ISR_DevC`) will be executed first?

```
.include "m128def.inc"
.def mpr = r16
START:
.org $0000
   RJMP INIT
.org $0002
   RCALL ISR_DevA
   RETI
.org $0008
   RCALL ISR_DevB
   RETI
.org $000C
   RCALL ISR_DevC
   RETI

INIT:
   …SP initialized…
   ldi    mpr, 0b10000011
   sts    EICRA, mpr
   ldi    mpr, 0b00001100
   out    EICRB, mpr
   ldi    mpr, ____________
   out    EIMSK, mpr
   ldi    mpr, $00
   out    DDRD, mpr
   ________________________(1)
   ldi    mpr,0b00001000
   ________________________(2)
   sei
      …
MAIN:
   {  …
      …do something…
   }

ISR_DevA:
   ________________________(3)
   ________________________(4)
   …
   …
```

```
    _______________________ (5)
    _______________________ (6)
    RET

ISR_DevB:
    {…
    …
    RET
    }
ISR_DevC:
    {…
    …
    RET
    }
```

Solution

(a) There are 8 external interrupt pins (INT7-INT0) in AVR. Which three interrupt pins are these I/O devices connected to?

This can be determined by looking at the addresses where RCALLs are located, i.e., addresses $0002, $0008, and $000C, which correspond to INT0, INT3, and INT5. These pins can also be recognized by values written into EICRA and EICRB. However, this may not always work since a pin can be set to "00" to indicate low-level trigger.

(b) Which I/O device's interrupt is detected on a falling edge?

Among the bits in EICRA and EICRB, bits 7 and 6 in EICRA is set to falling edge (i.e., 10). Therefore, DevB (INT3) is set to detect an interrupt on a falling edge.

(c) The interrupt pins referred to in part (a) are connected to two of the 7 ports (PORTA-PORTG) in AVR. Which ports are they?

INT7-INT0 are spread between PORTD (INT3-0) and PORTE (INT7-4).

(d) There are important instructions missing in the initialization portion (i.e., INIT:) of the code (which is underlined). Fill in the missing instructions in lines 1-2 so that the above code will work correctly. Note that DevB requires the pull-up resistor to be activated.

Since PORTE also needs to be setup for input, we need to have the following instructions:

```
out     DDRE, mpr            (1)
```

Since INT3 is set to detect on a falling edge, this pin needs to set to input with High-Z. Thus, we need to have the following instructions:

```
out     PORTD, mpr      (2)
```

(e) Suppose DevA requires that no interrupts can be generated while it is being serviced. Fill in lines 3-4 and 5-6 with the necessary code to mask out all other interrupts at the beginning of `ISR_DevA` and then reset the EIMSK so that latching of interrupts from the three devices can resume just before returning from `ISR_DevA`.

We need to clear EIMSK so that no interrupt will be latched onto EIFR. This is done by

```
ldi     mpr, 0b00000000 (3)
out     EIMSK, mpr      (4)
```

Then, we need to reset EIMSK so that the processor can continue to detect interrupts from the three I/O devices. This is achieved by

```
ldi     mpr, 0b00101001 (5)
out     EIMSK, mpr      (6)
```

(f) Suppose interrupts are detected from all three interrupt pins at the same time. Which subroutine (ISR_DevA, ISR_DevB, or ISR_DevC) will be executed first?

The priority is based on the address of RCALL with the lowest address having the highest priority. Thus, ISR_DevA will be called first.

[25 pts]
2- Consider the following AVR code segment:

```
.include "m128def.inc"
.def mpr = r16
.def zero = r17
.equ BASEADDR = $01F0

START:
.org $0000
   RJMP INIT
.org $000A
   RCALL ISR
   RETI
.org $0046
INIT:
      CLR   zero
      ...             ;Set up stack pointer, I/O, and Interrupts

ISR:
    ____________________________(1)
    ____________________________(2)
    ____________________________(3)
    ____________________________(4)
    ____________________________(5)
    ____________________________(6)
    ____________________________(7)
    ____________________________(8)
```

The eight lines of code for the sub-routine ISR shown above should function as follows:
1. Initialize X-pointer to BASEADDR
2. Read the input from Port D.
3. Add the input to the X-pointer
4. Load the value pointed to by X
5. Output that value to Port B
6. Go back to whatever it was doing before the interrupt occurred

Write the code for lines (1)-(8) so that the subroutine ISR works properly. Assume the subroutine INIT has already set up the stack, I/O, and interrupts.

Solution

```
ISR:
   LDI     XL, LOW(BASEADDR)
   LDI     XH, HIGH(BASEADDR)
   IN      mpr, PIND
```

```
ADD     XL, mpr
ADC     XH, zero
LD      mpr, X
OUT     PORTB, mpr
RET
```

[25 pts]

3- Consider the `WAIT` subroutine for Tekbot that waits for 1 sec. and returns. Rewrite the `WAIT` subroutine so that it waits for 1 sec. using the 16-bit Timer/Counter1. Assume that the system clock frequency is 16 MHz and the timer is operating under Normal mode. This is done by doing the following:

(1) Timer/Counter1 is initialized to operate in Normal mode.

(2) The `WAIT` subroutine loads the proper value into TCNT1 and waits until TOV1 is set. Once TOV1 is set, it is cleared and the `WAIT` subroutine returns.

Use the skeleton code shown below:

```
.include “m128def.inc”
.def mpr = r16
…
.ORG     $0000
    RJMP Initialize
.ORG     $0046           ; End of interrupt vectors
Initialize:
    …
    …Your code goes here…
    …

WAIT:
    …
    …Your code goes here…
    …
    RET
```

Solution

The first thing that needs to be done is to calculate the *value* to be loaded onto Timer/Counter1. This is done by evaluating the following equation:

value = 65535 – (1 sec/(prescale x 62.5 ns)) = 65535 – (16,000,000/prescale)

We want to use a prescale value that would lead to the highest resolution (i.e., lowest prescale value) and yet satisfy the above equation, thus prescale = 256. This leads to *value* = 3035.

Obviously, there are many ways to write this code, but here is one possibility:

```
.include “m128def.inc”
.def mpr = r16
…
.ORG     $0000
    RJMP Initialize
.ORG     $0046           ; End of interrupt vectors
Initialize:
    LDI  mpr, 0b00000100 ; Set prescalar to 256
    OUT  TCCR1B, mpr     ; By default this operates in Normal mode
…
```

```
…
WAIT:
    LDI  mpr, high(3035) ; Load the value for delay
    OUT  TCNT1H, mpr     ;
    LDI  mpr, low(3035)  ;
    OUT  TCNT1L, mpr     ;
LOOP:
    IN   mpr, TIFR       ; Read in TOV1
    ANDI mpr, 0b00000100 ; Check if TOV1 is set
    BREQ LOOP            ; Loop if TOV1 not set
    LDI  mpr, 0b00000100 ; Reset TOV1
    OUT  TIFR, mpr       ; Note – write 1 to reset
    RET
    …
```

The initialization part of the code sets the prescalar 256, i.e., CS12 = 1, CS11 = 0, and CS10 = 0. Note that the Normal mode of operation does not have to be explicitly configured since the Waveform Generation Mode bits are all 0's at reset, i.e., WGM13-10 = 0000 (Normal mode).

The WAIT routine first sets the Timer/Counter1 to *value* = 3035. Note that to do a 16-bit write, the high byte (TCNT1H) must be written before the low byte (TCNT1L) is written.

Finally, TOV1 is tested to see if it is set. If it is set, the loop exits and TOV1 is reset and the WAIT subroutine returns. Otherwise, the loop continues until TOV1 is set.

[25 pts]

4- Consider the AVR code segment shown below that performs initialization and receive routines for USART1 that continuously receives data from a transmitter. Write and explain the instructions in lines (1)-(16) necessary to make this code work properly. More specifically,
   (a) Fill in lines 1-4 with the necessary code to initialize the stack pointer.
   (b) Fill in lines 5-6 with the necessary code to configure RXD1 (Port D, pin 2).
   (c) Fill in lines 7-10 with the necessary code to set the Baud rate at 9,600 bps using double data rate.
   (d) Fill in lines 11-12 with the necessary code to enable the receiver and Receive Complete interrupt.
   (e) Fill in lines 13-14 to set the frame format to be asynchronous with 8 data bits, 2 stop bits, and even parity.
   (f) Fill in lines 15-16 to receive data and store it into a buffer.

```
.include "m128def.inc"
.def  mpr = r16

.ORG  $0000
      RJMP initUSART1;

.ORG  $003C
      RCALL RECEIVE_DATA;
      RETI

.ORG  $0046
initUSART1:
      ; Initialize Stack Pointer
      ________________________(1)
      ________________________(2)
      ________________________(3)
      ________________________(4)
      ; Configure Port D, pin 2
      ________________________(5)
      ________________________(6)
      ; Set baud rate at 9600bps, double data rate
      ________________________(7)
      ________________________(8)
      ________________________(9)
      ________________________(10)
      ; Enable receiver and Receive Complete interrupt
      ________________________(11)
      ________________________(12)
      ; Set frame format: 8 data, 2 stop bits, asynchronous, even parity
      ________________________(13)
      ________________________(14)

MAIN:
      LDI   XH, high(Buffer)
      LDI   XL, low(Buffer)
LOOP:
      RJMP  LOOP                    ; Wait for data
…

RECEIVE_DATA:
      push  mpr                     ; Save mpr
      ; Receive data and store in buffer
      ________________________(15)
      ________________________(16)
      pop   mpr                     ; Restore mpr
      ret                           ; Return
Buffer:
      .BYTE 100                     ; 100 byte buffer
```

Solution

Here is the code required to make this code work properly.

Lines 1-4
Sets the SP to point to the very last location in the SRAM.

Lines 5-6
RXD1, which is Port D pin 2, is set for input.

Lines 7-10
We need to first calculate the UBBR value, which is given by UBRR = (16MHz/(8x9600)) – 1 = 207. This value can then be stored into UBRR1H and UBRR1L in parts using high() and low() functions. Note that since our calculated UBRR value is less than 256, we only need to explicitly write to UBRR1L.

Lines 11-12
Sets RXEN1 and RXCIE1 bits in UCSR1B. Note that `sts` has to be used because UCSR1B is in the extended I/O space.

Lines 13-14
8-bit (UCSZ12:0 = 011), 2 stop bits (USBS1 = 1), even parity (UPM11:0 = 10), and asynchronous (UMSEL1 = 0).

Lines 15-16
Waits until Receive Complete interrupt occurs, and then moves the receive data to buffer.

```
.include    "m128def.inc"
.def  mpr = r16
.equ  BotID = $66
initUSART1:
; Initialize Stack Pointer
      ldi   mpr, high(RAMEND)       (1)
      out   SPH, mpr                (2)
      ldi   mpr, low(RAMEND)        (3)
      out   SPL, mpr                (4)
;Configure Port D, pin 2
      ldi   mpr, 0b00000000         (5)
      out   DDRD, mpr               (6)
;Set baud rate at 9600bps, double data rate
      ldi   mpr, (1<<U2X1)          (7)
      sts   UCSR1A, mpr             (8)
      ldi   mpr, low(207)           (9)
      sts   UBRR1L, mpr             (10)
```

```
; Enable transmitter and Receive Complete interrupt
      ldi   mpr, (1<<RXEN1|1<<RXCIE1)          (11)
      sts   UCSR1B, mpr                  (12)
; Set frame format: 8 data, 2 stop bits, asynchronous, even parity
      ldi   mpr, (0<<UMSEL1|1<<USBS1|1<<UPM11|0<<UPM10|1<<UCSZ11|1<<UCSZ10) (13)
      sts   UCSR1C, mpr                   (14)

MAIN:
      LDI   XH, high(Buffer)
      LDI   XL, low(Buffer)
LOOP:
      RJMP  LOOP                         ; Wait for data
…

RECEIVE_DATA:
      push  mpr                          ; Save mpr
      ; Receive data and store in buffer
      lds   mpr, UDR1                    (15)
      ST    X+, mpr                      (16)
      pop   mpr                          ; Restore mpr
      ret                                ; Return
Buffer:
      .BYTE 100                          ; 100-byte buffer
```

ECE 375
Computer Organization and Assembly Language Programming
Fall 2015
Solutions Set #4

The following questions are based on the enhanced AVR datapath (see Figures 8.24 and 8.26 in the text). The microoperations for the Fetch cycle are shown below.

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

[25 pts]
1- Consider the implementation of the `LD Rd,X+` (*Load Indirect and Post-Increment*) 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.
   Note that this instruction takes two execute cycles (EX1 and EX2). Control signals for the Fetch cycle are given below.

<u>Solution</u>
(a) Here is the sequence of microoperation for the `LD Rd,X+` instruction.

EX1: DMAR ← Xh:XL, Xh:Xl ← Xh:Xl + 1
EX2: Rd ← M[DMAR]

(b) The following shows the control signals and the RAL output.

Control Signals	IF	`LD Rd, X+`	
		EX1	EX2
MJ	0	x	x
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	x	1	x
Adder_f	xx	01	xx
Inc_Dec	x	x	x
MH	x	0	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:
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 to DMAR by setting MH to 0. At the same time, X is incremented by one by the Address Adder (via MUXG) by setting Adder_f to 01, and then written back to the Register File as 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. All other control signals can be don't cares except DM_w, which needs to be set to 0 so that the memory is not overwritten, and IR_en, PC_en, PCh_en, PCl_en, and SP_en, which are all set to 0's to prevent IR, PC, and SP, respectively, from being overwritten. The RAL output for rA and rB are set to Xh and Xl, respectively, so that the upper and lower bytes of X can be read from the register file. This is also the case for wA and wB since the updated value of X needs to be written back.

EX2:
The content of DMAR is routed through MUXE and used as an address to load the operand from Data Memory. The loaded operand is routed through MUXB and MUXC to the inB of the register file and written by setting RF_wB to 1. All other control signals can be don't cares except RF_wA, PC_en, PCh_en, PCl_en, and SP_en, which need to be set to 0 to prevent the Register File, 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. The RAL output for wB has to be set to Rd because the loaded value from memory has to be written to the destination register.

[25 pts]
2- Consider the implementation of the `POP Rd` (Pop Register from Stack) instruction on the enhanced AVR datapath.
   (a) List and explain the sequence of microoperations required to implement `POP Rd`.
   (b) List and explain the control signals and the Register Address Logic (RAL) output for the `POP Rd` 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

Control Signals	IF	`POP Rd`	
		EX1	EX2
MJ	0	x	x
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	1	0
DEMUX	x	x	x
MA	x	x	x
MB	x	x	1
ALU_f	xxxx	xxxx	xxxx
MC	xx	xx	00
RF_wA	0	0	0
RF_wB	0	0	1
MD	x	x	x
ME	x	x	0
DM_r	x	x	1
DM_w	0	0	0
MF	x	x	x
MG	x	x	x
Adder_f	xx	xx	xx
Inc_Dec	x	0	x

RAL Output	`POP Rd`	
	EX1	EX2
wA	x	x
wB	x	Rd
rA	x	x
rB	x	x

MH	x	x	x
MI	x	x	x

Solution

(a) The microoperations required are given below:

EX1: SP ← SP + 1
EX2: Rd ← M[SP]

(b) The control signals and RAL output requirements for the `POP Rd` instruction are shown below:

EX1:
First, SP needs to be incremented to point to the top of the stack. This is done by setting Inc_Dec to 0 and SP_en to 1. All other control signals can be don't cares except for RF_wA, RF_wB, and DM_w, which all need to be set to 0 to prevent Register File and Data Memory from being updated. In addition, IR_en, PC_en, PCh_en, PCl_en, which are set to 0 to prevent IR and PC from being overwritten.

EX2:
The value on the top of the stack is read from Data Memory by setting ME to 0 and DM_r to 1. Then, the read value is written to the Register File by setting MB to 1, MC to 00, and RF_wB to 1. Meanwhile, the RAL output for wB needs to be set to Rd so that the popped value is written to the proper location in Register File. All other control signals can be don't cares except RF_wA, PC_en, PCh_en, PCl_en, DM_w, and SP_en, which are set to 0 to prevent Register File, PC, Data Memory, and SP from being overwritten. Note that IR_en can be "don't care" since this is the last execute cycle and IR register will be overwritten in the fetch (i.e., next) cycle.

[25 pts]
3- Consider the implementation of the `RCALL k` (Relative Subroutine Call) instruction on the enhanced AVR datapath.
   (a) List and explain the sequence of microoperations required to implement `RCALL k`.
   (b) List and explain the control signals and the Register Address Logic (RAL) output for the `RCALL k` 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)
EX1: M[SP] ←RARl, SP ← SP – 1
EX2: M[SP] ←RARh, SP ← SP – 1, PC ← PC + 1 + se k12
(b)

Control Signals	IF	RCALL k	
		EX1	EX2
MJ	0	x	1
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	0
MG	x	x	0
Adder_f	xx	xx	00
Inc_Dec	x	1	1
MH	x	x	x
MI	x	0	1

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

EX1:
SP provides the address for the Data Memory by setting ME to 0, and then RARl is selected by setting MI to 0 and written to the Data Memory by setting MD to 0 and DM_w to 1. At the same, time, SP is decremented by setting Inc/Dec to 1 and latched on to the SP by setting SP_en to 1. All other control signals can be "don't cares" except IR_en, RF_wA, RF_wB, and NPC_en, which need to be set to 0 to prevent IR register, Register File, and RAR register respectively, from being overwritten. Note that PC_en can be don't care since PC will be overwritten in EX2. [Note: Do we need to mention why DM_r must be 0 here and in EX2?]

EX2:
SP provides the address for the Data Memory by setting ME to 0, and then RARh is selected by setting MI to 1 and written to the Data Memory by setting MD to 0 and DM_w to 1. SP is also decremented by setting Inc/Dec to 1 and latched on to the SP by setting SP_en to 1. At the same time, se k12 (i.e., sign-extended 12-bit k) is added to NPC (which is PC + 1) by setting Adder_f to 00, generating the target address PC+1+ se k12. Then, PC+1+ se k12 is routed and latched onto the PC by setting MJ to 1 and PC_en to 1. All other control signals can be don't cares except RF_wA and RF_wB, which need to be set to 0 to prevent the Register File from being overwritten. Note that IR_en can be "don't care" since this is the last execute cycle and IR register will be overwritten in the fetch (i.e., next) cycle. Also note that NPC_en can be "don't care" because the content of RAR (as well as NPC) will not change until the end of the cycle.

[25 pts]

4- 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.

Solution

(a) Here is the sequence of microoperations for the `LPM` instruction.

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

(b) The following shows the control signals and the RAL output.

Control Signals	IF	LPM EX1	LPM EX2	LPM EX3
MJ	0	x	x	x
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	xxxx
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	x	1	x	x
Adder_f	xx	11	xx	xx
Inc_Dec	x	x	x	x
MH	x	x	x	x
MI	x	x	x	x

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

EX1:
The Zh and Zl registers are read from the Register File by providing Zh and Zl to rA and rB, respectively. Zh:Zl is then latched onto the PMAR register. This is done by routing through the Address Adder by setting MG to 1 and Adder_f to 11. All other control signals can be "don't cares" except RF_wA and RF_wB to prevent the register file from being updated. In addition, IR_en, DM_w, PC_en, PCh_en, PCl_en, and SP_en need to be set to 0's to prevent IR register, Data Memory, PC register, and SP register, respectively, from being overwritten. The RAL output for rA and rB are set to Zh and Zl, respectively, so that the upper and lower bytes of Z can be read from the register file.

EX2:
The Program Memory is read based on PMAR by setting ML to 1, and then the value read is latched onto MDR. All other control signals can be don't cares except RF_wA/RF_wB, IR_en, DM_w, PC_en, PCh_en, PCl_en, and SP_en, which need to be set to 0's to prevent the register file, IR register, Data Memory, PC register, and SP register, respectively, from being overwritten. Finally, RAL output can be all don't cares because the register file is not used.

EX3:
The content of MDR is written back to R0 in the register file by setting MC to 10, wB to R0, and RF_wB to 1. All other control signals can be don't cares except RF_wA, DM_w, PC_en, PCh_en, PCl_en, and SP_en, which need to be set to 0's to prevent Register File, Data Memory, 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. The RAL output for wB has to be set to Rd (which happens to be 0) because the loaded value from Program Memory has to be written to a destination register.