[20 pts]

1- Consider the internal structure of the pseudo-CPU augmented with a Stack Pointer (SP) and a 16-bit Temporary (TEMP) register. 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 19 microoperations.* 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
Cycle 1: MAR ← PC;
Cycle 2: MDR ← M[MAR], PC ← PC + 1 ; Get the high byte of the opcode and increment PC
Cycle 3: IR(15…8) ← MDR
Cycle 4: MAR ← PC;
Cycle 5: MDR ← M[MAR], PC ← PC + 1 ; Get the low byte of the opcode and increment PC
Cycle 6: IR(7…0) ← MDR ; At this point, CU knows this is a CALL
; 16-bit target address is fetched
Cycle 7: MAR ← PC;
Cycle 8: MDR ← M[MAR], PC ← PC + 1
Cycle 9: TEMP(15…8) ← MDR
Cycle 10: MAR ← PC;
Cycle 11: MDR ← M[MAR], PC ← PC + 1
Cycle 12: TEMP(7…0) ← MDR

; The return address is pushed onto the stack
Cycle 13: MDR ← PC(7…0)
Cycle 14: MAR ← SP
Cycle 15: M[MAR] ← MDR, SP ← SP - 1 ; Push the lower byte of return address onto stack
Cycle 16: MDR ← PC(15…8)
Cycle 17: MAR ← SP
Cycle 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
Cycle 19: PC ← TEMP ; Put the TA into PC (points to address of subroutine)

[20 pts]
2- 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:

Fetch cycle
Cycle 1: MAR ← PC;
Cycle 2: MDR ← M[MAR], PC ← PC+1 ; Get the high byte of the instruction and increment PC
Cycle 3: IR ← MDR ; At this point, CU knows this is adiw
Cycle 4: MAR ← PC;
Cycle 5: MDR ← M[MAR], PC ← PC+1 ; Get the low byte of the instruction and increment PC

Execute cycle
Cycle 6: AC ← R30
Cycle 7: AC ← AC + MDR ; Add 32 to ZL
Cycle 8: R30 ← AC ; Write back to register file
Cycle 9: AC ← R31
Cycle 10: If (C=1) then AC ← AC +1 ; Increment ZH if there was a carry
Cycle 11: R31 ← AC ; Write it back to register file

[20 pts]
3- Consider the following AVR assembly code that performs 16-bit by 16-bit multiplication (with some information missing). Assume the data memory locations $0100$ through $0107$ initially have the following values:

<table>
<thead>
<tr>
<th>Data Memory</th>
<th>Address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>0101</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>0102</td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>0103</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>0104</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0105</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0106</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>0107</td>
<td>00</td>
<td></td>
</tr>
</tbody>
</table>

.include "m128def.inc" ; Include definition file
.def rlo = r0 ; Low byte of MUL result
.def rhi = r1 ; High byte of MUL result
.def zero = r2 ; Zero register
.def A = r3 ; An operand
.def B = r4 ; Another operand
.def oloop = r17 ; Outer Loop Counter
.def iloop = r18 ; Inner Loop Counter
.org $0000
1. rjmp INIT
.org $0046
2. INIT: clr zero ; Set zero register to zero
3. MAIN: ______________ ; Load low byte
4. ______________ ; Load high byte
5. ______________ ; Load low byte
6. ______________ ; Load high byte
7. ldi oloop, ___ ; Load counter
8. MUL16_OLOOP: ______________ ; Load low byte
9. ______________ ; Load high byte
10. ldi iloop, 2 ; Load counter
11. MUL16_ILOOP: ldi A, X+ ; Get byte of A operand
12. ldi B, Y ; Get byte of B operand
13. mul A, B ; Multiply A and B
14. ldi A, Z+ ; Get a result byte from memory
15. ldi B, Z+ ; Get the next result byte from memory
16. add rlo, A ; rlo <= rlo + A
17. adc rhi, B ; rhi <= rhi + B + carry
18. ldi A, Z ; Get a third byte from the result
19.  
   adc  A, zero  ; Add carry to A
20.  
   st   Z, A    ; Store third byte to memory
21.  
   st   -Z, rhi ; Store second byte to memory
22.  
   st   -Z, rlo ; Store first byte to memory
23.  
   adiw  ZH:ZL, 1 ; Z <= Z + 1
24.  
   dec  iloop  ; Decrement counter
25.  
   brne  MUL16_ILOOP ; Loop if iLoop != 0
26.  
   sbiw  ZH:ZL, 1 ; Z <= Z - 1
27.  
   adiw  YH:YL, 1 ; Y <= Y + 1
28.  
   dec  oloop  ; Decrement counter
29.  
   brne  MUL16_OLOOP ; Loop if oLoop != 0
30.  
   Done:  
   rjmp  Done
       .org  $0100
   addrA: .byte  2
   addrB: .byte  2
   LAddrP: .byte  4

(a) Show the code needed to initialized the X (lines 8-9), Y (lines 3-4), and Z (lines 5-6) pointers to point to labels addrA, addrB, and LAddrP, respectively.

(b) What are the two 16-bit values (in hexadecimal) being multiplied?

(c) What are the contents of memory locations pointed to by LAddrP, LAddrP+1, LAddrP+2, and LAddrP+3 after the loop MUL16_ILOOP (lines 11-25) completes for the first time (i.e., the 1st iteration)?

(d) What are the contents of memory locations pointed to by LAddrP, LAddrP+1, LAddrP+2, and LAddrP+3 after the loop MUL16_ILOOP (lines 11-25) completes for the second time (i.e., the 2nd iteration)?

(e) What is the immediate value needed in the instruction in line 7 for this program to work properly?

Solution:

This code performs the following:

(a) Show the code needed to initialized the X (lines 8-9), Y (lines 3-4), and Z (lines 5-6) pointers to point to labels addrA, addrB, and LAddrP, respectively.

(b) What are the two 16-bit values (in hexadecimal) being multiplied?

The figure below shows where labels addrA, addrB, and LAddrP are pointing to in data memory after executing lines 3-9.

<table>
<thead>
<tr>
<th>Data Memory</th>
<th>Address</th>
<th>content</th>
</tr>
</thead>
<tbody>
<tr>
<td>addrA:</td>
<td>0100</td>
<td>03</td>
</tr>
<tr>
<td></td>
<td>0101</td>
<td>02</td>
</tr>
<tr>
<td>addrB:</td>
<td>0102</td>
<td>0C</td>
</tr>
<tr>
<td></td>
<td>0103</td>
<td>01</td>
</tr>
</tbody>
</table>
Since X points to $0100$ and Y points to $0102$, the two values are either $0203$ and $010C$ or $0302$ and $0C01$. However, since the first multiplication (line 13) performed in the inner loop is $03 \times 0C$, the two values are $0203$ and $010C$.

(c) What are the contents of memory locations pointed to by $\text{LAddrP}$, $\text{LAddrP+1}$, $\text{LAddrP+2}$, and $\text{LAddrP+3}$ after the loop $\text{MUL16_ILOOP}$ (lines 11-25) completes for the first time (i.e., the 1st iteration)?

Lines 11-13 result in the following:

\[
\begin{array}{cc}
03 \\
00 & \text{x} & 0C \\
00 & 24 \\
\end{array}
\]

(rhi)(rlo)

Lines 14-19 result in

\[
\begin{array}{cc}
00 & \text{(rhi)(rlo)} \\
00 & 24 \\
\end{array}
\]

\[
\begin{array}{cc}
00 & \text{(B)} & \text{(A)} \\
00 & 00 \\
\end{array}
\]

\[
\begin{array}{cc}
00 & \text{(A)} & \text{(rhi)(rlo)} \\
00 & 24 \\
\end{array}
\]

Then, lines 20-22 stores these three bytes in

\[
\begin{array}{cc}
0104 & 24 \\
0105 & 00 \\
0106 & 00 \\
\end{array}
\]

(d) What are the contents of memory locations pointed to by $\text{LAddrP}$, $\text{LAddrP+1}$, $\text{LAddrP+2}$, and $\text{LAddrP+3}$ after the loop $\text{MUL16_ILOOP}$ (lines 11-25) completes for the second time (i.e., the 2nd iteration)?

Lines 11-13 result in the following:

\[
\begin{array}{cc}
02 \\
00 & \text{x} & 0C \\
00 & 18 \\
\end{array}
\]

(rhi)(rlo)

Lines 14-19 result in

\[
\begin{array}{cc}
00 & \text{(rhi)(rlo)} \\
00 & 18 \\
\end{array}
\]

\[
\begin{array}{cc}
00 & \text{(B)} & \text{(A)} \\
00 & 00 \\
\end{array}
\]

\[
\begin{array}{cc}
00 & \text{(A)} & \text{(rhi)(rlo)} \\
00 & 18 \\
\end{array}
\]

Then, lines 20-22 stores these three bytes in

\[
\begin{array}{cc}
0104 & 24 \\
0105 & 18 \\
0106 & 00 \\
\end{array}
\]
(c) What is the immediate value needed in the instruction in line 7 for this program to work properly?

Since the outer-loop needs to be repeated for the higher byte, the instruction required is

7. \texttt{ldi \ loop, 2} ; Load counter

[20 pts]

4. For the following AVR code, determine the machine code for each instruction in the program code shown below. Some of the machine codes have already been determined.

\begin{verbatim}
.include "m128def.inc"
.def mpr = r16
.def count = r17

.ORG $0000
START: \texttt{RJMP INIT}
0000: 1100 kkkk kkkk kkkk
 .ORG $0002
  RCALL ISR
0002: 1101 kkkk kkkk kkkk
  RETI
0003: 1001 0101 0001 1000

INIT: {
  \ldots
  Interrupt \ldots
  Initialization \ldots
  Code \ldots
  \ldots
}

  \texttt{LDI XH, high(CTR)}
000B: 1110 KKKK dddd KKKK

  \texttt{LDI XL, low(CTR)}
000C: 1110 KKKK dddd KKKK

  \texttt{LDI YH, high(DATA)}
000D: 1110 KKKK dddd KKKK

  \texttt{LDI YL, low(DATA)}
000E: 1110 KKKK dddd KKKK

WAIT: \texttt{RJMP WAIT}
000F: 1100 kkkk kkkk kkkk

 .ORG $100F
ISR: \texttt{IN mpr, PINA}
100F: 1011 0AAAd dddd AAAA

  \texttt{ST Y+, mpr}
1010: 1001 001r rrrr 1001

  \texttt{INC count}
1011: 1001 010d dddd 0011

  \texttt{ST X+, count}
1012: 1001 001r rrrr 1101

  \texttt{RET}
1013: 1001 0101 0000 1000

.DSEG
 .ORG $0100
CTR: .BYTE 1
DATA: .BYTE 256
\end{verbatim}

\begin{center}
\begin{tabular}{llc}
\textbf{Program Address} & \textbf{Binary} \\
\hline
\texttt{START: RJMP INIT} & 0000: 1100 kkkk kkkk kkkk \\
\texttt{.ORG $0002} & \ldots \\
\texttt{RCALL ISR} & 0002: 1101 kkkk kkkk kkkk \\
\texttt{RETI} & 0003: 1001 0101 0001 1000 \\
\texttt{INIT: \{} & 0004: \ldots \\
\texttt{\ldots} & \ldots \\
\texttt{Interrupt} & \ldots \\
\texttt{Initialization} & \ldots \\
\texttt{Code} & \ldots \\
\texttt{\ldots} & \ldots \\
\texttt{\}} & \ldots \\
\texttt{LDI XH, high(CTR)} & 000B: 1110 0000 0000 1011 \\
\end{tabular}
\end{center}

Solution:

\begin{verbatim}
.include "m128def.inc"
.def mpr = r16
.def count = r17

.ORG $0000
START: RJMP INIT
0000: 1100 0000 0000 0011
 .ORG $0002
  RCALL ISR
0002: 1101 \ldots \ldots \ldots
  RETI
0003: 1001 0101 0001 1000

INIT: {
  \ldots
  Interrupt \ldots
  Initialization \ldots
  Code \ldots
  \ldots
}

  \texttt{LDI XH, high(CTR)}
000B: 1110 0000 1011 0001
\end{verbatim}
Program Address $0000:
Since INIT is located at $0004 and RJMP is located at $0000, the displacement for the PC-relative jump is $0004-$0001=$0003. Thus, kkkk kkkk kkkk =0000 0000 0011.

Program Address $0002:
Since ISR is located at $100F and RCALL is located at $0002, the displacement for the PC-relative jump is $100F-$0003=-$0001. Thus, kkkk kkkk kkkk =1111 1111 1111, which is 2’s complement (or negative) of 1.

Program Address $000B:
XH = R29, which is 1 1101. For LDI, the MSB of 1 is implied and thus only the lower 4 bit are encoded. Therefore, dddd = 1011. The upper (left) and lower (right) KKKKs are determined by the 8-bit immediate value of high(CTR). Since CTR is located at $0100, the higher portion of $0100 is $01. Therefore, the upper KKKK is 0000 and the lower KKKK is 0001.

Program Address $000C:
YL = R28, which is 1 1100. For LDI, the MSB of 1 is implied and thus only the lower 4 bit are encoded. Therefore, dddd = 1100. The upper (left) and lower (right) KKKKs are determined by the 8-bit immediate value of low(DATA). Since DATA is located at $0101, the lower portion of $0101 is $00. Therefore, the upper KKKK is 0000 and the lower KKKK is 0000.

Program Address $000E:
Since WAIT is located at $000F and RJMP is located at $000F, the displacement for the PC-relative jump is $000F-$0010 =$-0001. Thus, kkkk kkkk kkkk =1111 1111 1111, which is 2’s complement (or negative) of 1.

Program Address $100F:
The I/O address of PINA is $19 and mpr is R16. Therefore, AAd dddd AAAA = 011 0000 1001.

Program Address $1010:
Since mpr is R16, r rrrr = 1 0000.
Program Address $1011:
Since count is R17, $ddd = 10001.

Program Address $1012:
Since count is R17, $rrr = 10001.

[20 pts]
5- Suppose the following array of numbers are stored in the Data Memory (represented in hexadecimal):

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:</td>
<td>01</td>
</tr>
<tr>
<td>0001:</td>
<td>BE</td>
</tr>
<tr>
<td>0002:</td>
<td>35</td>
</tr>
<tr>
<td>0003:</td>
<td>EC</td>
</tr>
<tr>
<td>0004:</td>
<td>48</td>
</tr>
<tr>
<td>0005:</td>
<td>2D</td>
</tr>
<tr>
<td>0006:</td>
<td>04</td>
</tr>
<tr>
<td>0007:</td>
<td>02</td>
</tr>
</tbody>
</table>

Using AVR assembly language, write a subroutine that (1) determines the smallest number among the 8 numbers stored in memory and (2) stores the smallest number in memory location $0008. Clearly comment and explain your code. Use the skeleton code shown below to implement your subroutine:

```
.ORG  $0046
.Initialize stack...
RCALL MIN
...
.ORG  $0060
MIN:
... ; Your code goes here
...
RET
```

**Solution:**

One possible code is shown below:

```
MIN:
LDI  XH, $00 ; Initialize the X register to point to
LDI  XL, $00 ; the beginning of the array
CLR  R18 ; Clear temporary register
LDI  R19, 8 ; initialize count
LOOP:
LD  R17, X+ ; Load the array value and post-increment X
CP  R17, R18 ; Compare loaded value with previous smallest value
BRGE SKIP ; If greater than equal to, then skip.
MOV  R18, R17 ; Else, move loaded value to temporary register
SKIP:
DEC  R19 ; Decrement count
BRNE LOOP ; Check if count has reached zero, if not jump to loop
DONE:
ST  X, R18 ; When done, store the smallest value in location $0008
RET ; Return from subroutine
```
Note that I did not use any registers R7-R0 (the registers I used were R17-R19). This is because registers R0-R7 are mapped to data memory locations $0000$-$0007$. If I had used these registers, I would have inadvertently overwritten some of these values in memory.

Another interesting aspect of this code is that the numbers in the memory can be unsigned or signed. BRGE instruction does a signed comparison and will treat numbers with MSB set as negative (2’s-complement) numbers. So this program would work if we meant to operate in signed numbers. If we meant to work with unsigned numbers, we would have to use BRMI (Branch if minus) instead of BRGE.