ADC - Add with Carry Adds two registers and the contents of the C flag and places the result in the destination register Rd. 0001 11rd dddd rrrr

ADD - Add without Carry Adds two registers without the C flag and places the result in the destination register Rd. Rd ← Rd + Rr. 0000 11rd dddd rrrr

ADIW - Add Immediate to Word Adds an immediate value (0-63) to a register pair and places the result in the register pair. This instruction operates on the upper four register pairs, and is well suited for operations on the pointer registers. This instruction is not available in all devices. Refer to the device specific instruction set summary. Rd+1:Rd ← Rd+1:Rd + K. 1001 0110 KKdd KKKK

BRCC - Branch if Carry Cleared Conditional relative branch. Tests the Carry flag (C) and branches relatively to PC if C is cleared. If C = 0 then PC <- PC + k + 1, else PC <- PC + 1. 1111 01kk kkkk k000

BRNE - Branch if Not Equal. Conditional relative branch. Tests the Zero flag (Z) and branches relatively to PC if Z is cleared. If Rd ≠ Rr (Z = 0) then PC ← PC + k + 1, else PC ← PC + 1. 1111 01kk kkkk k001

CALL - Long Call to a Subroutine. Calls to a subroutine within the entire program memory. The return address (to the instruction after the CALL) will be stored onto the stack. 1001 010k kkkk 111k kkkk kkkk kkkk kkkk

CP- Compare. This instruction performs a compare between two registers Rd and Rr. None of the registers are changed. All conditional branches can be used after this instruction. CP Rd,Rr 0 ≤ d ≤ 31, 0 ≤ r ≤ 31 PC ← PC + 1. 0001 01rd dddd rrrr

LD - Load Indirect from data space to Register using Index X. Loads one byte indirect from the data space to a register. LD Rd, X . 1001 000d dddd 1100

LDI - Load Immediate. Loads an 8 bit constant directly to register 16 to 31. LDI Rd,K. 1110 KKKK dddd KKKK

MOV - Copy Register. This instruction makes a copy of one register into another. The source register Rr is left unchanged, while the destination register Rd is loaded with a copy of Rr. MOV Rd,Rr. 0010 11rd dddd rrrr

MUL- Multiply Unsigned. This instruction performs 8-bit × 8-bit → 16-bit unsigned multiplication. result placed in r1, r0 (HL). MUL Rd,Rr. 1001 11rd dddd rrrr

NEG- Two's Complement. Replaces the contents of register Rd with its two's complement; the value $80 is left unchanged. NEG Rd. 1001 010d dddd 0001

SBC- Subtract with Carry. Subtracts two registers and subtracts with the C flag and places the result in the destination register Rd. SBC Rd,Rr. 0000 10rd dddd rrrr

ST - Store Indirect From Register to data space using Index X. Stores one byte indirect from a register to data space. ST X+, r26. (i) 1001 001r rrrr 1100

SUB- Subtract without Carry. Subtracts two registers and places the result in the destination register Rd. SUB Rd,Rr. 0001 10rd dddd rrrr

ICALL - Indirect Call to Subroutine. ndirect call of a subroutine pointed to by the Z (16 bits) pointer register in the register file. 1001 0101 0000 1001

IJMP - Indirect Jump. Indirect jump to the address pointed to by the Z (16 bits) pointer register in the register file.

JMP - Jump. Jump to an address within the entire 4M (words) program memory. JMP k. 1001 010k kkkk 110k kkkk kkkk kkkk kkkk

LDS - Load Direct from data space Loads one byte from the data space to a register. LDS Rd,k. 1001 000d dddd 0000 kkkk kkkk kkkk kkkk

LPM - Load Program Memory. Loads one byte pointed to by the Z register into the destination register Rd. LPM Rd, Z. 1001 000d dddd 0100

```
STA -(x) ; M(x) ← M(x)-1, M(M(x)) ← AC,
Fetch Cycle
Step 1: MAR ← PC;
Step 2: MDR ← M(MAR), PC ← PC+1
Step 3: IR ← MDR opcode , MAR← MDR address ; Read inst. & increment PC
Execute Cycle
Step 1: MDR ← M(MAR) ; Get EA+1 from memory (i.e., M(x))
Step 2: TEMP ← AC ; Save AC to TEMP
Step 3: AC ← MDR ; Decrement EA+1
Step 4: AC ← AC -1 ;
Step 5: MDR ← AC ; Store EA into M(x)
Step 6: M(MAR) ← MDR ;
Step 7: AC ← TEMP ; Restore AC
Step 8: MAR ← MDR ; Have MAR point to EA
Step 9: MDR ← AC ; Store content of AC into M(M(x))=M(EA)
Step 10: M(MAR) ← MDR ;
```

Consider the following hypothetical 1-address assembly instruction called “Store Accumulator Indirect with Pre-Decrement” of the form STA -(x) ; M(x) ← M(x)-1, M(M(x)) ← AC, 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 operation code 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 cycles for the above STA -(x) instruction. Your solution should result in minimum number of microoperations. Assume PC is currently pointing to the STA instruction and only PC and AC have the capability to increment/decrement itself. This instruction does not modify the original content of the AC. Fetch cycle is given below

Things to remember:

Stack starts high and decrements as stuff is placed on it.

Arithmetic: converting dec to binary: do you need x to get y? yes=1, no = 0.

Dec to Hex: divide number by 16 and that is high byte, low byte = original number - the closest you got.

Two’s complement:

Two's complement is the way every computer I know of chooses to represent integers. To get the two's complement negative notation of an integer, you write out the number in binary. You then invert the digits, and add one to the result.

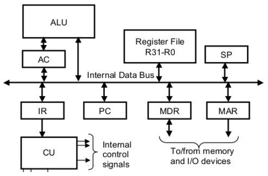
```
adiw ZH:ZL,32
Fetch cycle
Step 1: MAR ← PC;
Step 2: MDR ← M(MAR), PC ← PC+1 ; Get the high byte of the instruction and increment PC
Step 3: IR ← MDR ; At this point, CU knows this is adiw
Step 4: MAR ← PC;
Step 5: MDR ← M(MAR), PC ← PC+1 ; Get the low byte of the instruction and increment PC
Execute cycle
Step 6: AC ← R30
Step 7: AC ← AC + MDR ; Add 32 to ZL
Step 8: R30 ← AC ; Add 32 to ZL
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
```

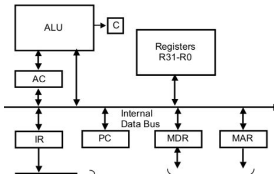
Consider the internal structure of the pseudo-CPU discussed in class augmented with a single-port register file (i.e., only one register value can be read at a time) 32 8-bit registers (R31-R0) and a carry bit (C-bit), which is set/reset after each arithmetic operation. Suppose the pseudo-CPU can be 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.

Decimal	Hex	Binary
0	0	00000000
1	1	00000001
2	2	00000010
3	3	00000011
4	4	00000100
5	5	00000101
6	6	00000110
7	7	00000111
8	8	00001000
9	9	00001001
10	a	00001010
11	b	00001011
12	c	00001100
13	d	00001101
14	e	00001110
15	f	00001111
16	10	00010000

```
ICALL
; Fetch cycle
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 an ICALL;
Execute cycle
; The return address is pushed onto the stack
Step 7: MDR ← PC(7...0)
Step 8: MAR ← SP
Step 9: M(MAR) ← MDR, SP ← SP-1 ; Push the lower byte of return address onto stack
Step 10: MDR ← PC(15...8)
Step 11: MAR ← SP
Step 12: M(MAR) ← MDR, SP ← SP-1 ; Push the higher byte of return address onto stack
: Put H and L target addresses of the Z register to the PC
Step 13: PC(15...8) ← R31 ; Put the H address of the target into PC
Step 14: PC(7...0) ← R30 ; Put the L address of the target into PC
```

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) containing 32 8-bit registers (R0-R31) and a Stack Pointer (SP). Suppose the pseudo-CPU can be used to implement the AVR instruction ICALL (Indirect Call to Subroutine) ICALL pushes the return address onto the stack and jumps to the 16-bit target address contained in the Z register. Give the sequence of microoperations required to Fetch and Execute AVR’s ICALL instruction. Your solutions should result in exactly 6 cycles for the fetch cycle and 8 cycles for the execute cycle. Assume the memory is organized into addressable bytes (i.e., each memory word is a byte), MDR register is 8-bit wide, and SP, PC, IR, and MAR 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 SP has the capability to increment/decrement itself. Clearly state any other assumptions made.





Assembler Directives

ORG: Tells the assembler where to put the instructions that follow it.

.ORG 0x37

EXIT: Tells the assembler to stop assembling the file

EQU: Assigns a value to a label.

Syntax: .EQU label = expression

.EQU io_offset = 0x23

BYTE: Reserves memory in data memory

Syntax: label: .BYTE expression

var: .BYTE 1

DB: Allows arbitrary bytes to be placed in the code.

Syntax: label: .DB expressionlist

consts: .DB 0, 255, 0b01010101, -128, 0xaa

Text: .DB “This is a text.”

DW: Allows 16-bit words to be placed into the code

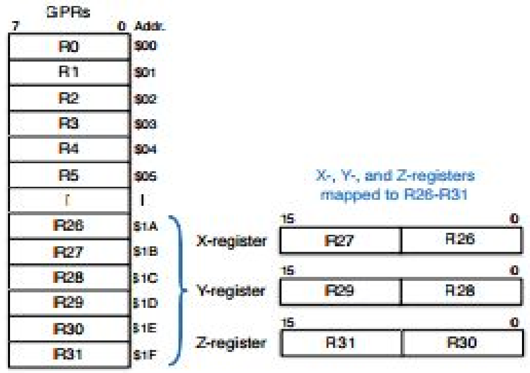
Syntax: label: .DW expressionlist

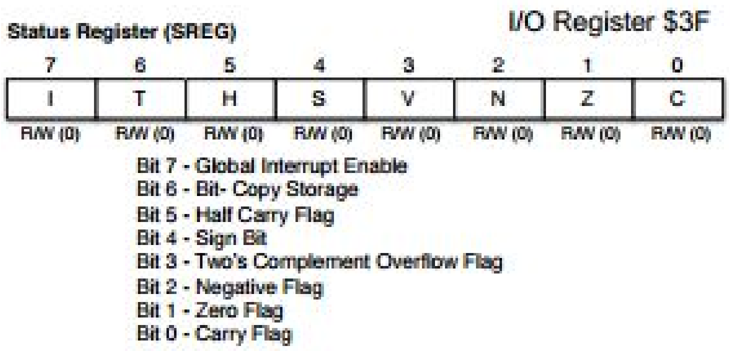
varlist: .DW 0,0xffff,0b1001110001010101,-32768,65535

INCLUDE: Tells the assembler to read from the specified file.

Syntax: .INCLUDE filename

.INCLUDE “m128def.inc”





Bit 7 - Global Interrupt Enable

Bit 6 - Bit- Copy Storage

Bit 5 - Half Carry Flag

Bit 4 - Sign Bit

Bit 3 - Two's Complement Overflow Flag

Bit 2 - Negative Flag

Bit 1 - Zero Flag

Bit 0 - Carry Flag

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. - EX: Rd - K

(b) List and explain the control signals and the Register Address Logic (RAL) output for the CPI Rd,K instruction.

EX: The content of Rd is read from the Register File by providing the register identifier Rd to rA. At the same time, the constant K from the instruction is routed (via the Alignment Unit) through MUXA by setting MA to 0. The ALU then performs a subtract operation (ALU_f = 0010), which then sets the appropriate condition flags (e.g., C, Z, N, V, and S flags in SREG). Note that the result of the ALU operation does not have to be stored back into the register file. All other control signals can be don't cares except DM_w, RF_wA, and RF_wB, which need to be set to 0's so that the memory and the register file are not updated, and PC_en (as well as PCh_en and PCl_en) and SP_en are set to 0's to prevent PC and SP from being overwritten. Note that IR_en can be "don't care" since this is the last (the only) execute cycle and the IR register will be overwritten in the fetch (i.e., next) cycle.

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+. - EX1: DMAR ¬ Yh:YL, Yh:Yl ¬ Yh:Yl + 1 EX2: Rd ¬ M[DMAR] (b) List and explain the control signals and the Register Address Logic (RAL) output for the LD Rd,Y+. Instruction.

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 (via MUXG) by setting Adder_f to 01, and then latched onto YH and YL (via MUXC) by setting both RF_wA and RF_wB to 1's and providing Yh and Yl 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, as well as PCh_en and 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 Yh and Yl, respectively, so that the upper and lower bytes of Y can be read from the register file. This is also the case for wA and wB since updated value of Y needs to be written back.

EX2: The content of DMAR is routed through MUXE and used to fetch the operand from Data Memory. The fetched 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 DM_w, which needs to be set to 0 so that the memory is not overwritten. PC_en (as well as PCh_en and PCl_en), SP_en, RF_wA, which all needs to be set to 0 to prevent the PC register, the SP register, and the register file, 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

Consider the 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. - EX1: SP ¬SP + 1, EX2: PCh ¬ M[SP], SP ¬ SP + 1, EX3: PCl ¬ M[SP]

(b) List and explain the control signals and the Register Address Logic (RAL) output for the RET instruction

EX1: The content of SP is routed to the Increment/Decrement Unit, and incremented by setting Inc_Dec to 0. The incremented SP is then relatched onto SP by setting SP_en to 1. All other control signals can be "don't cares" except RF_wA/RF_wB, DM_w, IR_en, and PC_en (as well as PCh_en and PCl_en), which all need to be set to 0's to prevent the register file, Data Memory, IR, and PC from being overwritten with unwanted values.

EX2: The content of SP is routed to the Increment/Decrement Unit, and incremented by setting Inc_Dec to 0. The incremented SP is then relatched onto SP by setting SP_en to 1. At the same time, the Data Memory location

pointed to by SP, i.e., M[SP], which is the higher 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. The read value is then routed through DEMUX to the upper

byte of PC, i.e., PCh by setting DEMUX to 1 and PCh_en to 1. All other control signals can be don't cares except RF_wA/RF_wB, DM_w, IR_en, and PC_en (as well as PCl_en), which all need to be set to 0's to prevent the

register file, Data Memory, IR, and PC from being overwritten with unwanted values.

EX3: The Data Memory location pointed to by SP, i.e., 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. The read value is then routed to DEMUX to the lower byte of PC, i.e., PCl, by setting DEMUX to 0 and PCl_en to 1. All other control signals can be don't cares except Sp_en, RF_wA/RF_wB, DM_w and PC_en (as well as PCh_en),, which all need to be set to 0's to prevent the SP, register file, Data Memory, and PC from being overwritten with unwanted values. 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.

Control Signals	IF	CPI
		EX
MJ	0	x
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	x	x
Adder_f	xx	xx
Inc_Dec	x	x
MH	x	x
MI	x	x

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

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

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

Control Signals	IF	LD Rd, Y+	
		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

Control Signals	IF	RCALL	
		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	01	01
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

Control Signals	IF	LPM		
		EX1	EX2	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	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	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	RET		
	EX1	EX2	EX3
wA	x	x	x
wB	x	x	x
rA	x	x	x
rB	x	x	x

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

Control Signals	IF	RET		
		EX1	EX2	EX3
MJ	0	x	x	x
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	xx	xx	xx	xx
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	x	x	x	x
Adder_f	xx	xx	xx	xx
Inc_Dec	x	0	0	x
MH	x	x	x	x
MI	x	x	x	x

Consider the implementation of the RCALL (Relative Call to Subroutine) instruction on the enhanced AVR datapath.

(a) List and explain the sequence of microoperations required to implement RCALL. - EX1: M[SP] ← RARl, SP ← SP - 1 EX2: M[SP] ← RARh, SP ← SP - 1, PC ← NPC + se k

(b) List and explain the control signals and the Register Address Logic (RAL) output for the RCALL instruction

EX1: SP provides the address for the Data Memory by setting ME to 0, and then RARl is written to the Data Memory by setting MI to 0, MD to 0 and DM_w to 1. At the same time, SP is decremented using the Increment/Decrement Unit by setting Inc_Dec to 1 and latched onto the SP by setting SP_en to 1. All other control signals can be "don't cares" except IR_en and NPC_en, which need to be set to 0's to prevent the IR register and the NPC register (as well as RAR) from being overwritten. Note that PC_en can be don't care since PC will be overwritten in EX2. Finally, RAL output can be all don't cares because the register file is not used.

EX2: SP provides the address for the Data Memory by setting ME to 0, and then RARh is written to the Data Memory by setting MI to 1, MD to 0 and DM_w to 1. At the same time, SP is decremented using the Increment/Decrement Unit by setting Inc_Dec to 1 and latched on to the SP by setting SP_en to 1. In addition, NPC and se K are added using the Address Adder by setting MG to 0, MF to 0, and Adder_f to 00. The resulting target address is latched onto the PC by setting MJ to 1and PC_en to 1. All other control signals can be don't cares. 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. Also, 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, so there is no danger of the return address in RAR being modified while the RARh is pushed onto the stack, and will be updated by the fetching of the instruction from the target anyway. Finally, RAL output can be all don't cares because the register file is not used.

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

(b) List and explain the control signals and the Register Address Logic (RAL) output for the LPM instruction.

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, 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, 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, rB to 00000002, and RF_wB to 1. All other control signals can be don't cares except DM_w, PC_en, and SP_en, which need to be set to 0's to prevent 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 memory has to be written to a destination register.

Consider the AVR assembly code shown below with its equivalent (partially completed) binaries on the right. Determine the values for

(a) KKKK dddd KKKK (@ address $0049)
(b) KKKK dddd KKKK (@ address $004B)
(c) kk kkkk k (@ address $005D)
(d) KKdd KKKK (@ address $005E)
(e) kkkk kkkk kkkk (@ address $0062)

```
                .org    $0000               Address          Binary
                rjmp    INIT                0000:      1100  kkkk   kkkk   kkkk
                .org    $0046               …                ‾
INIT:           clr     r2                  0046:      0010  01rd   dddd   rrrr
MAIN:           ldi     YL, low(addrB)      0047:      1110  KKKK   dddd   KKKK
                ldi     YH, high(addrB)     0048:      1110  KKKK   dddd   KKKK
                ldi     ZL, low(LAddrP)     0049:      1110  KKKK   dddd   KKKK
                ldi     ZH, high(LAddrP)    004A:      1110  KKKK   dddd   KKKK
                ldi     r17, 2              004B:      1110  KKKK   dddd   KKKK
MUL16_OLOOP:    ldi     XL, low(addrA)      004C:      1110  KKKK   dddd   KKKK
                ldi     XH, high(addrA)     004D:      1110  KKKK   dddd   KKKK
                ldi     r18, 2              004E:      1110  KKKK   dddd   KKKK
MUL16_ILOOP:    ld      r3, X+              004F:      1001  000d   dddd   1101
                ld      r4, Y               0050:      1000  000d   dddd   1000
                mul     r3,r4               0051:      1001  11rd   dddd   rrrr
                ld      r3, Z+              0052:      1001  000d   dddd   0001
                ld      r4, Z+              0053:      1001  000d   dddd   0001
                add     r0, r3              0054:      0000  11rd   dddd   rrrr
                adc     r1, r4              0055:      0001  11rd   dddd   rrrr
                ld      r3, Z               0056:      1000  000d   dddd   0000
                adc     r3, r2              0057:      0001  11rd   dddd   rrrr
                st      Z, r3               0058:      1000  001d   dddd   0000
                st      -Z, r1              0059:      1001  001d   dddd   0010
                st      -Z, r0              005A:      1001  001d   dddd   0010
                adiw    ZH:ZL, 1            005B:      1001  0110   KKdd   KKKK
                dec     r18                 005C:      1001  010d   dddd   1010
                brne    MUL16_ILOOP         005D:      1111  01kk   kkkk   k001
                sbiw    ZH:ZL, 1            005E:      1001  0111   KKdd   KKKK
                adiw    YH:YL, 1            005F:      1001  0111   KKdd   KKKK
                dec     r17                 0060:      1001  010d   dddd   1010
                brne    MUL16_OLOOP         0061:      1111  01kk   kkkk   k001
Done:           rjmp    Done                0062:      1100  kkkk   kkkk   kkkk
                .org    $0100
addrA:          .byte   2
addrB:          .byte   2
LAddrP:         .byte   4
```

The ZL is register is r30 = 1 1110. The BYTE directive allocates a byte of memory in Data Memory address space. `.ORG $0100` followed by `addrA: .byte 2` and `addrB: .byte 2` dictates that `LAddrP` is $0104. Thus, `low(LAddrP)` represents low byte of address `LAddrP`, which is $04. Therefore, KKKK dddd KKKK = 0000 1110 0100

rd = r17 = 1 0001 and the immediate value is $02, thus KKKK dddd KKKK = 0000 0001 0010

The target address of `brne MUL16_ILOOP` is $004F, and `brne MUL16_ILOOP` is at $005D, therefore, PC+1 i at $005E. Thus $004F_{16} – 005E_{16} = -000F_{16} = -0001111_2 = 1110001_2$.

Thus, kk kkkk k = 11 1000 1

`sbiw` instruction works on upper four register pairs, i.e., r25:r24 (dd=00), r27:r26 (dd=01), r29:r28, (dd=10) and r31:r30 (dd=11). The immediate value is 1. Therefore, KKdd KKKK = 0011 0001

The target address of `rjmp Done` is $0062, and `rjmp Done` is at 0062, thus, PC+1 is at $0063. Therefore, $0062_{16} – 0063_{16} = -0001_{16} = -000000000001_2 = 111111111111_2$.

Thus, kkkk kkkk kkkk = 1111 1111 1111

BIT AND BIT-TEST INSTRUCTIONS

LSL	Rd	Logical Shift Left	Rd(n+1) ← Rd(n),Rd(0) ← 0,C ← Rd(7)
LSR	Rd	Logical Shift Right	Rd(n) ← Rd(n+1),Rd(7) ← 0,C ← Rd(0)
ROL	Rd	Rotate Left Through Carry	Rd(0) ← C,Rd(n+1) ← Rd(n),C ← Rd(7)
ROR	Rd	Rotate Right Through Carry	Rd(7) ← C,Rd(n) ← Rd(n+1),C ← Rd(0)
ASR	Rd	Arithmetic Shift Right	Rd(n) ← Rd(n+1), n=0..6
SWAP	Rd	Swap Nibbles	Rd(3..0) ↔ Rd(7..4)
BSET	s	Flag Set	SREG(s) ← 1
BCLR	s	Flag Clear	SREG(s) ← 0
SBI	P, b	Set Bit in I/O Register	I/O(P, b) ← 1
CBI	P, b	Clear Bit in I/O Register	I/O(P, b) ← 0
BST	Rr, b	Bit Store from Register to T	T ← Rr(b)
BLD	Rd, b	Bit load from T to Register	Rd(b) ← T

```
.def mpr = r16                          ; Multi-purpose register
.def count = r17                        ; Assume R17 is initially 0
.ORG $0000
START: RJMP INIT
.ORG $0002
       RCALL ISR
       RETI
INIT:  LDI   mpr, 0b00000011            ; Sets Input Sense Control for pin INT0
       STS   EICRA, mpr                 ; to detect an interrupt on a rising edge
       LDI   mpr, 0b00000001            ; Enables interrupt for pin INT0
       OUT   EIMSK, mpr                 ;
       LDI   mpr, $00                   ; Set Port A Direction Register for input
  OUT  DDRA, mpr                        ;
       SEI                              ; Turn on interrupts
       LDI   XH, high(CTR)
       LDI   XL, low(CTR)
       LDI   YH, high(DATA)
       LDI   YL, low(DATA)
WAIT:  RJMP WAIT
.ORG   0x100F
ISR:   IN    mpr, PINA
       ST    Y+, mpr
       INC   count
       ST    X, count
       RET
.DSEG
CTR:   .BYTE 1
DATA:  .BYTE 256
```

Write an AVR assembly code that waits for 1 sec using the 8-bit Timer/Counter0 with the system clock frequency of 16 MHz operating under Normal mode. This is done by doing the following:

(1) Timer/Counter0 is initialized to count for 10 ms and then interrupts on an overflow;

(2) The main part of the program simply loops, and for each iteration, a check is made to see if the loop has reach 100 iterations; and

(3) On each interrupt, Timer/Counter0 is reloaded to interrupt again in 10 ms.

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 = 255 – (10 ms/(prescale x 62.5 ns)) = 255 – (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 = 1024. This leads to value = 99. Obviously, there are many ways to write this code, but here is one possibility:

```
.include "m128def.inc"
.def mpr = r16
.def counter = r17
…
.ORG $0000
RJMP Initialize
.ORG $0020 ; Timer/Counter0 overflow interrupt vector
RCALL Reload_counter
RETI
.ORG $0046 ; End of interrupt vectors
Initialize:
LDI mpr, 0b00000100 ; Enable interrupt on Timer/Counter0 overflow
OUT TIMSK, mpr
SEI ; Enable global interrupt
LDI mpr, 0b00000111 ; Set prescalar to 1024
OUT TCCR0, mpr ;
LDI mpr, 99 ; Load the value for delay
OUT TCNT0, mpr ;
LDI counter, 100 ; Set timer
LOOP:
CPI counter, 0 ; Repeat for 100 times
BRNE LOOP
…
Reload_counter:
PUSH mpr
LDI mpr, 16 ;
OUT TCNT0, mpr ; Load the value for delay
DEC counter
POP mpr
RET
```

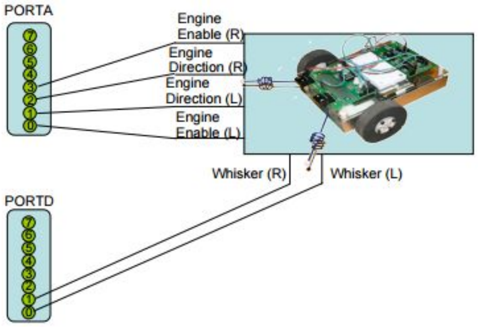
The initialization part of the code first enables Timer/Counter0 overflow interrupt and the global interrupt. Then, the prescalar is set to 1024, i.e., CS02 = 1, CS01 = 1, and CS00 = 1. Note that the Normal mode of operation does not have to be explicitly configured since the Waveform Generation Mode bits are all 0s at reset, i.e., WGM13-10 = 0000 (Normal mode). The next part sets the Timer/Counter0 to value = 99. Once the Timer/Counter0 is set, the program enters a loop waiting for a Timer/Counter0 overflow interrupt to occur. In addition, counter is checked to see if it has reached zero. Finally, the Reload_counter routine decrements the counter, reloads the value, and returns.

Consider the AVR code segment shown below that initializes and handles interrupts. Descriptions of Data Directional Register (DDRx), External Interrupt Control Registers (EICRA & EICRB), and External Interrupt Mask Register (EIMSK) are given on the following page.

(a) Explain in words what the code accomplishes when it is executed. That is, explain what it does and how it does it. - This code reads an 8-bit value latched on to Port A when an interrupt occurs from INT0. It then stores it to memory starting at address DATA then increments a count, which will be stored in memory location at CTR.

(b) Write and explain the interrupt initialization code (lines (1)-(7)) necessary to make the interrupt service routine (starting at ISR:) work properly. More specifically,

```
.include "m128def.inc"
.def mpr = r16 ; Multi-purpose register
.def count = r17 ; Assume R17 is initially 0
.ORG $0000
START: RJMP INIT
.ORG $0002
RCALL ISR
RETI
INIT: LDI mpr, 0b00000011 ; Sets Input Sense Control for pin INT0
STS EICRA, mpr ; to detect an interrupt on a rising edge
LDI mpr, 0b00000001 ; Enables interrupt for pin INT0
OUT EIMSK, mpr ;
LDI mpr, $00 ; Set Port A Direction Register for input
OUT DDRA, mpr ;
SEI ; Turn on interrupts
LDI XH, high(CTR)
LDI XL, low(CTR)
LDI YH, high(DATA)
LDI YL, low(DATA)
WAIT: RJMP WAIT
.ORG 0x100F
ISR: IN mpr, PINA
ST Y+, mpr
INC count
ST X, count
RET
.DSEG
CTR: .BYTE 1
DATA: .BYTE 256
```



Write a subroutine initUSART1 to configure ATmega128 USART1 to operate as a transmitter and sends a data every time USART1 Data Register Empty interrupt occurs. The transmitter operates with he following settings: 8 data bits, 2 stop bits, and even parity, 9,600 Baud rate, Transmitter enabled, Normal asynchronous mode operation, Interrupt enabled, Assume the system clock is 16 MHz. The skeleton code is shown below: There are many ways to write this code but here is one possible code for initUSART1.

```
initUSART1:
; Port D set up – pin3 output
ldi mpr, 0b00001000 ; Configure USART1 (Port D, pin 3)
out DDRD, mpr ; Set pin direction to output
; Set Baud rate
ldi mpr, 103 ; Set baud rate to 9,600 with f = 16 MHz
sts UBRR1L, mpr ; UBRR1H already initialized to $00
; Enable transmitter and interrupt
ldi mpr, (1<<TXEN1|1<<UDRIE1) ; Enable Transmitter and interrupt
sts UCSR1B, mpr
; Set asynchronous mode and frame format
ldi mpr, (1<<USBS1|1<<UPM11|1<<UCSZ11|1<<UCSZ10)
sts UCSR1C, mpr ; UCSR1C in extended I/O space, use sts
ret
```

The first two instructions configure Pin 3, Port D for output since the USART1 is acting as a transmitter. The next two instructions set the Baud rate. This is done by calculating the UBRR value, which is (16MHz/(16x9600))-1 = 103, and then writing it into the UBRR1L register. The UBRR1H register was not written to since upper byte is $00. The next two instructions enable the transmitter and the interrupt, which is done by setting the 3rd-bit (i.e., TXEN1) and the 5th-bit (i.e., UDRIE1) of UCSR1B. The next pair of instructions sets it to the asynchronous mode, which is done by setting the 6th bit (i.e., UMSEL1) of UCSR1C to 0. Note that 0<<UMSEL1 is not necessary since it is already initialized to 0 on reset. This is also the case for 0<<UCSZ12 and 0<<UPM10. The frame format with 8 data bits, 2 stop bits, and even parity is set by selecting UCSZ12:0 to be 011, UPM11:0 to be 10, and USBS1 to 1. These bits are configured using (1<<USBS1|1<<UPM11|0<<UPM10|0<<UCSZ12|1<<UCSZ11|1<<UCSZ10) Also note that sts is used because UCSR1C is in the extended I/O space. Here is a possible code for SendData, which is called when USART1 Data Register Empty interrupt occurs.

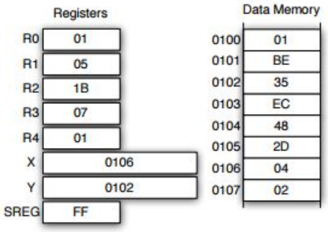
```
SendData:
ld r17, X+ : Assume X points to the data to be transmitted
sts UDR1, r17 ; Move data to Transmit Data Buffer
Ret
```

Consider the AVR code segment shown below that initializes I/O and interrupts for Tekbot shown below.

```
.include "m128def.inc"
.def mpr = r16
.org $0000
rjmp INIT
…
INIT: ldi mpr, 0b00001111 (1) ; Set DDRA to control engine
out DDRA, mpr (2) ;
ldi mpr, 0b00000000 (3) ; Set DDRD to detect whiskers
out DDRD, mpr (4) ;
ldi mpr, 0b00000011 (5) ; Enable pull-up resisters for L/R whiskers
out PORTD, mpr (6) ;
ldi mpr, 0b00001010 (7) ; Set EICR to detect on falling edge
sts EICRA, mpr (8) ;
ldi mpr, 0b00000011 (9) ; Set EIMSK
out EIMSK, mpr (10) ;
sei ; Turn on interrupts
```

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)   sbiw   XH:XL, 2
(ii)  ldi    r27, 85
(iii) ror    r2
(iv)  adc    r2, r1
(v)   sts    $0007, r28
```



Solution:

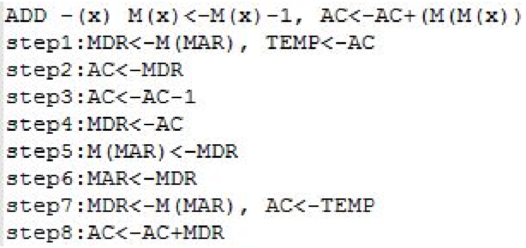
(i) Since X is $0106, subtracting 2 results in $0104. Thus, X changes to $0104.

(ii) R27 changes to $\$55_{16}$. Thus, X changes to 5506_{16}

(iii) R2 is 00011011_2, thus rotating it right through the carry, which is 1, results in 10001101_2=$8D_{16}$. Thus, R2 changes to $8D.

(iv) Status Register (SREG) indicates FF, thus C-bit (LSB) is set.
Thus, $1B_{16}$+ 05_{16} + 01(carry) = 21_{16}
R2 changes to $21

(v) Since R28 is lower byte of Y registers, r28 = $02, and thus M[0007]=$02.



```
ADD -(x) M(x)<-M(x)-1, AC<-AC+(M(M(x))
step1:MDR<-M(MAR), TEMP<-AC
step2:AC<-MDR
step3:AC<-AC-1
step4:MDR<-AC
step5:M(MAR)<-MDR
step6:MAR<-MDR
step7:MDR<-M(MAR), AC<-TEMP
step8:AC<-AC+MDR
```

Consider the internal structure of the pseudo-CPU discussed in class augmented with with a *single-port register file* (i.e., only one register value can be read at a time) containing 32 8-bit registers (R0-R31) and a Stack Pointer (SP) register. Suppose the very simple CPU can be used to implement the AVR instruction **RET** (Return from Subroutine) with the format shown below:

```
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 RET

Step 7:  SP ← SP+1
Step 8:  MAR ← SP
Step 9:   MDR ← M(MAR), SP ← SP+1    ; Pop the higher byte of return address from the stack
Step 10: PC(15…8) ← MDR
Step 11: MAR ← SP
Step 12: MDR ← M(MAR),               ; Pop the lower byte of return address from the stack
Step 13: PC(7…0) ← MDR

Goto fetch and Execute cycle
```