# SAMPLE EXAM QUESTIONS.



im watching *game of thrones*

*> There will be one question you’ve never seen before and we haven’t covered it in lectures, but i’m sure you’ll be fine*

*> I am a good lecturer. I am with you.*

*> so amaze. very doge. much wow.*

*mfw sri will take the iron throne*

<http://gifatron.com/wp-content/uploads/2013/03/tyrion-eyebrows-game-of-thrones.gif>

<http://www.cse.unsw.edu.au/~cs2121/Tutorials/SampleExamQuestions.pdf>

Bookmarks:

[Extra Qns (by Oliver Tan)](#h.4u46x12cq1um)

[Sample Final Exam](#h.jsma8rbo9jtx)

**Main differences between AVR and ARM:** //sdouble check pls, am not good with computer

//ARMv7 used

> AVR has a 2 stage single level pipeline (fetch, execute), ARM has three stage pipeline (fetch, decode, execute)

> AVR has three main memory spaces (data, program, EEPROM) (Harvard), ARM stores instructions and data in the same space (Von Neumann/Princeton (ARM9 uses Harvard though..)).

> AVR registers 8 bit [32 General Registers, 64+416 I/O registers]

> ARM registers 32 bit [37 Registers, 31 General, 6 Status]

> AVR I/O is mapped directly to memory spaces. If we wish to interface with the devices, we must use special I/O instructions. ARM is similar, but no separate I/O address space. Interfaced with as if they are any other part of memory. I think?

ARMv7 and Atmel AVR similarities:

* RISC-based Harvard architecture
* Lower-complexity instruction sets
* Support for CPU power saving
* Similar addressing modes
* Status register and flagging functionality

ARM unique features:

* 32-bit architecture
* Dynamic-stage pipelining
* Three operand encoding
* Optimized program SREG, allows for conditional execution
* ARM Cortex-A9 ALU allows for multiple-bit shifting
* Full floating point operation support (w/ VFPv3 add-on)
* Functionality able to be extended with coprocessor support

AVR unique features:

* 8-bit architecture
* Simpler two-stage pipelining (fetch and execute)
* Extensive I/O register (memory-mapped I/O) and peripheral support

Applications:

* ARM is suited to large-scaled embedded applications

-> Higher functionality allows for greater complexity

-> Higher manufacturing cost

-> Suitable for industrial automation and handheld smart phones

* AVR is suited to smaller projects

-> Easy to learn ISA with sufficient basic functionality -> smaller applications

-> Low cost and power consumption architecture

-> Suitable for home appliances and energy-saving lighting

**1. Convert the following numbers from the original base to the specified base:**

a) 12310 -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_2

0111 10112

b) 101012 -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_10

2110

c) 108410 -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_16

43C16

d) A516 -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_10

16510

e) 110010012 ->\_\_\_\_\_\_\_\_\_\_\_\_\_\_16

C916

f) 2D516 -> \_\_\_\_\_\_\_\_\_\_\_\_\_\_2

0010 1101 01012

**2. What is the result of the following calculations?**

**a)** 1395 + 4988 (base 16) |add 2 numbers together, if >15,next number +1. start from 8+5 ,then 9+8 ...

5D1D16

**b)** 11001001 + 00101101 (base 2)

111101102

**c)** A41 – 560 (base 16)

(1)4E116

**d)** 11001 – 011 (base 2)

25 - 3 = 22 -> 101102

3 = 00011, -3 in 2’s comp is 11101

11101 + 11001 = 10110 = 22

**4. What number does 10010010 represent as an unsigned number? What does it \represent in 2’s complement notation?**

Unsigned: just add the bits up. 2+16+128 = 146

Twos complement: the sign bit being set means it's negative, so invert the digits and add one

10010010 -> 01101101 + 1 -> 01101110 -> (2+4+8+32+64) \* -1 = -110.

Alternatively do 2^8 - 146 (unsigned) = 256 - 146 = 110 then make it negative -110

**5. In 2’s complement addition, 11011011 + 01100000 = 00111011. Was there a 2’s complement overflow? Why? What do the values in this sum represent?**

There was no 2’s complement overflow. Since the most significegative, and likewise it will be positive for the second. For the equation above, this will be something in the form (-) + (+) =ant bit in the first number is a 1, we know that this is n (+) (similar to how -5 + 6 = 1). This is not a 2’s complement overflow.

A 2’s complement overflow would be 01111111 + 00000001 = 10000000; which is 127 + 1 = -128. You can think of this in a number line, with the far left being -128, 0 being the centre, and the far right being 127. That addition is simply crossing that 0 boundary. A 2’s complement underflow would therefore crossing the -128 boundary to the positive boundary by a minus operation.

There’s no 2’s complement overflow as the carry into the most significant bit is equal to the carry out of the MSB.

Values:

-(37) + 96 = 59

Also: (You don’t get overflows with one positive and one negative number. Overflows are neg+neg -> pos or pos+pos -> neg)

**6. What is the difference between performing 2’s complement addition and unsigned addition in the AVR processor?**

The actual process of adding the bits together is the same, however in 2’s complement addition, the N flag in the status register can be set, indicating that the result would be negative if the numbers were signed and represented as 2’s complement.

Check V flag for 2’s complement overflow, C flag for unsigned ‘overflow’/carry.

The S flag is the exclusive OR of N and V flags, and may be set in 2’s complement addition.

**7. Represent the following numbers in IEEE 754 32-bit floating point notation:**

a) 1.5

1.5 = 1 x 20 + 1 x 2-1

= 1.12

So sign = 0 (positive)

Exponent = 0 (+ 127 bias)

Mantissa = 1 (With 22 trailing 0s)

Therefore 1.5 in Floating point representation is 0 01111111 10000000000000000000000

Sign Exponent Mantissa

= 3FC0 000016

b) 1084

1084 = 100001111002

= 1.0000111100 x 210

Sign = 0 (positive) d

Exponent = 10 (+ 127 bias = 137 =10001001)

Mantissa = 00001111 2(followed by 15 0s)

Therefore 1084 in floating point representation is 0 10001001 00001111000000000000000

Sign Exponent Mantissa

=4487 800016

c) –1

Sign = 1 (negative)

Exponent = 0 (+ 127 bias = 01111111)

Mantissa = 23 0s

Therefore -1 in floating point representation is 1 01111111 00000000000000000000000

Sign Exponent Mantissa

=BF80 000016

d) –13.75

-13.75 = 1x2^3 + 1x2^2 + 0x2^1 + 1x2^0 + 1x2^-1 + 1x2^-2

= 1101.11

= 1.10111 x 2^3

Sign = 1 (negative)

Exponent = 3 (+ 127 bias = 130 = 10000010 base 2)

Mantissa = 10111 Followed by 18 0s

so -13.75 = 1 10000010 101110000000000000000000

Sign Exponent Mantissa

=C15C 000016

**8. What does the following IEEE 754 FP number represent:**

0 1000 0001 110 0000 0000 0000 0000 0000

Sign Exponent Mantissa

We know the number is positive from the unset sign bit

Exponent = 129 (-127 bias = 2)

So the number is 1.11 x 2^2

= 1112 = 710

**9. Encode the following instructions into Atmel AVR machine code:**

a) ldi r18, 127

Format for ldi is:

Ldi Rd, K K = 127 -> in binary = 0111 1111

Rd = r18. However as you can see in the opcode for ldi we only have 4 bits (d). That means only 16 references can be made. e.g. 0000 = first reference and 1111 (15) = last reference. Therefore we have a range of r16 - r31. (Why they picked the upper registers, was a design thing). Therefore R18 = 2nd reference = 0010. .. Same goes for everything else.

1110 KKKK dddd KKKK

Answer: 1110 0111 0010 1111

b) mov r18, r2

Format for mov is:

mov Rd, Rr

0010 11rd dddd rrrr

Answer: 0010 1101 0010 0010

c) lds r2, 0xABCD

Format for lds is:

lds Rd, k

1001 000d dddd 0000 kkkk kkkk kkkk kkkk

Answer: 1001 0000 0010 0000 1010 1011 1100 1101

**10. How many bits are needed to address:**

a) 16 32-bit general purpose registers?

16 registers -> 2^4 registers: 4 bits

b) a memory space of 65536 bytes (assume byte addressing)?

65536 bytes -> 2^16 byte sized registers: 16 bits

c) a memory space of 65536 32-bit words (assume byte addressing)?

Sri confirmed 18 bits is the correct answer.

65536 32-bit words -> 2^16 \* 2^2 bytes 18 bits

The answer is 100% 18 bits. we’re using byte addressing on 32-bit words (4 bytes long) so we need 2 extra bits to reference the extra stuff.

So if we want to refer to each of the 65536 words we need

16 bits  
 But actually we want to refer to each byte of these  
 (each of 4 bytes(32-bits) takes 2 bits to refer to)

+2 bits

= 18 bits

|  |  |
| --- | --- |
| m Byte Address | 8-bit (1 byte wide) |
| 1 | 32-bit data  (4 bytes) |
| 2 |
| 3 |
| 4 |

**11. What do the following letters in a typical status register stand for and how are they generated?**

**.**

a) Z

Zero Flag; indicates a zero result in an arithmetic or logic operation. 1: zero. 0: Non-zero

b) C

Carry flag; for addition it is the carry of the most significant bit. For subtraction (X-Y), it indicates whether |X|>|Y|. If |Y|>|X|, C is set, otherwise not set.

c) V

Two’s complement overflow flag; used in twos complement arithmetic. Set when a twos complement result overflows. *V = C XOR (Carry in to MSB)*

*Other way to explain: Set when carry\_in != carry\_out for the MSB*

*NOTE: Flag will still be set/cleared when performing unsigned arithmetic (the computer can’t tell the difference). Use the correct comparisons to check these flags and interpret the result.*

d) N

Negative flag; The Most Significant Bit of the result of last arithmetic operation

e) S

Sign Flag; Exclusive OR (XOR) between the Negative Flag N and the Two’s Complement Overflow Flag V

**12. What is the main difference between the memory models of Princeton (von Neumann) and Harvard architectures?**

Princeton / von Neumann: one memory space is shared between everything. (Allowing for chips that can load custom code, and modify their code while running). Princeton is cheaper and simpler to manufacture than Harvard

Harvard: Program memory (instructions) and data memory are stored in separate address spaces (allowing for faster reads, better security)

AVR is (modified) Harvard

/// WE DONT NEED THIS RIGHT>? (It was in the first assignment so I would be surprised if we needed it)

**13. Based on the below, what is the 32-bit word stored at the memory address 0x00000100 in the options below**

Memory address Data

0x00000100 0xAF

0x00000101 0x1B

0x00000102 0xC2

0x00000103 0x05

a) big-endian machine?

0xAF 1B C2 05 [do we convert to decimal? [\xAF\x1B\xC2\x05]

b) little-endian machine? (Starts with least significant byte @ smallest address)

0x05 C2 1B AF [\x05\xC2\x1B\xAF]

**14. Can you design an 8-bit instruction format that can allow 4 2-operand instructions for a machine with 8 registers?**

Yes,

4 opcodes - 2bits (00 - 11)

8 registers - 3 bits (000 - 111)

opcode Rr Rk

oo rrr kkk

ie. oorr rkkk

**15. What do these notations mean in AVR assembly programming? Where are they used?**

**a) .def d) .dseg g) .dw**

a. .def: Define a symbolic name on a register/Gives a register an alias

d. .dseg: Data segment (Data memory)

g. .dw: define constant word(s), little endian rule is used (program memory)

Store word(16-bit) constants in program memory.

**b) .set e) .org h) .byte**

b. .set: Set a symbol to an expression (we can change it later). Define symbol for value .set input = 5

e. .org: set program origin (e.g. if we .dseg then .org 0x100, we set .dseg to start at 0x100)

h. .byte: reserve byte size to variable (in data space)

**c) .cseg f) .db i) .equ**

c. .cseg: Code segment (Program memory)

f. .db: Define constant byte(s) (in program memory)

i. .equ: Set a symbol equal to an expression which is un-redefinable afterwards.

**16. Where are the functions low() and high() utilised? Load -200 into a two byte number.**

low and high load the low or high byte (respectively) of a 16-bit constant. This allows 16-bit numbers to be easily loaded into a pair of AVR’s 8-bit registers. For example to load -200 into r17:r16,  
  
ldi r16, low(-200)  
ldi r17, high(-200)

**17. What are the differences between Macros and Functions? In what circumstances are each of them appropriate, and when should each be avoided? Write a Macro called Invert to invert the value of a register**

Macros take arguments and are expanded inline. Functions are called, utilising the stack to store the location of the next instruction before jumping to another location to execute the function and then jumping back to the instruction stored on the stack. Macros are faster, as no jumping or stack operations are required, but inflate the memory size of the program. Functions have the overhead of setting up the stack and jumping, but allow the reuse of code, conserving memory.

When you call a macro, the AVR preprocessor simply replaces the call with the code in the macro definition (similar to #define in C), so calling macros in multiple locations can increase the code size significantly. (Flash memory is limited so if program is too big then program cannot be loaded!)

.macro invert

mov r16, @0  
 ser r17  
 eor r16, r17  
 mov @0, r16

.endmacro

alternatively,

.macro invert

com @0

.endmacro

**18. What are word addressable and byte addressable? Explain them with examples using AVR memories.**

AVR **program memory** Is word addressable. this means that address 0 specifies the first word, address 1 the second and so on. each program memory address refers to a pair of bytes.

* PC and Labels in AVR Studio refer to the word address
* LPM instruction requires the byte addressSo program memory is both word and byte addressable depending on context.   
    
  AVR SRAM and EEPROM are **byte addressable**, each byte has its own address.
* Word addressing: each address points to a distinct word.

Byte addressing: each address points to a distinct byte.

**19. Consider the following AVR assembly code segment and fill the initialization part?**

**.dseg**

**array: .byte 20**

**.cseg**

**data: .dw 0x1234**

**// Initialize the X pointer with array**

ldi XH, high(array)

ldi XL, low(array)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**// Initialize the Z pointer with data**

ldi ZH, high(data<<1)

ldi ZL, low(data<<1)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**20. What are little endian and big endian representations ? Which endian is used in AVR?**

Little endian is where the least significant byte of a word is located at the smallest memory address, whereas in big endian, the most significant byte is at the lowest memory address.

x86 (PCs) is a well-known little endian architecture.

AVR is little endian (for program memory)

e.g. 0x123ABC

Little Endian Big Endian

1000 BC 1000 12

1001 3A 1001 3A

1002 12 1002 BC

AVR is a little endian architecture.

* didnt sri say avr doesn’t have an endian l0l
* I remember sri said that AVR can use either endian, it depends how we wish to set it up

Big endian is used when interacting with the network.

*FYI: Little endian is what pretty much all computers you will interact with use. You reverse the order of the bytes, eg 0x12345678 becomes 0x78 0x56 0x34 0x12.*

*This actually makes sense because when you're storing numbers that're smaller than the architecture length, eg storing a 16 bit number on a 32 bit machine, you can still index the memory from the start of the field. eg*

*[\_\_ \_\_ \_\_ \_\_] being a 32 bit field, if you store [12 34 \_\_ \_\_] you can still refer to it with the address 0x10000000 or wherever it's up to; rather than having to add on something to the address.*

*Big endian only exists because people go "oh but my numbers are backwards in memory"*

*[well okay apparently it's because intel liked little endian and since it was decided in the 70s people stuck with it]*

**21. Identify the errors in the following instructions,**

a) ldi r1,18

Cannot use r1 with ldi instruction, only r16 - r31.

b) cp r16,‘L’

CP compares two registers and does not compare with a constant

c) ldi zh, high(0x3476) => Word Addressable

ldi ZL, low(0x3476 << 1)

ldi ZH, high(0x3476 << 1)

d) ldi r40, 23

There is no register 40. LOL

e) brge loop => for both unsigned numbers

brge is only used with signed[?] numbers? Will work for unsigned numbers that don’t have the last bit set though.

In this case brsh would be used.

*Yes, this is correct*

f) brlo end => for both signed numbers

brlo is unsigned, must use brlt.

.

**22. Write AVR assembly code segments for the following scenarios,**

a) Initialize an array A of size 20 (each element is one byte) with values ranging from 1 to 20.

A: .db 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

or

(assuming A has been reserved in program memory)

ldi XL, low(A<<1)

ldi XH, high(A<<1)

.def counter = r16

ldi counter, 1

loop:

cpi counter, 21

breq end

st X+, counter

inc counter

rjmp loop

end:

rjmp end

b) Initialize an array B of size 20 (each element is two bytes) with values ranging from -1 to -20.

B: .dw -1, -2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -14, -15, -16, -17, -18, -19, -20

!!! each element is two bytes i.e. a word, therefore you must store the two’s complement value across the whole word!

i.e. for -1 -> 1111 1111, so the above .dw defining works, but examples below don’t

or ?

ldi XL, low(A)

ldi XH, high(A)

.def counter = r16

clr counter

loop:

dec counter

cpi counter, -21

breq end

st X+, counter

inc X ← you can’t inc X, its made up of two registers. adiw Xh:Xl, 1

rjmp loop

end:

rjmp end0

or ?

.dseg

B: .byte 40

main:

ldi XH, high(B)

ldi XL, low(B)

clr r16

clr r17

loop:

dec r16

st X+, r17

st X+, r16

cpi r16, -20

brne loop

end:

rjmp end

c) Add the arrays A and B together and store the result into an array C.

(code below works, tested with AVR. Need to load data from program memory into data memory then add them together. Previous code used ld to try and load from program memory….)

.include "m2560def.inc"

.dseg

A: .byte 20

B: .byte 40

C: .byte 40

.cseg

rjmp initA

A\_P: .db 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

B\_P: .dw -1,-2, -3, -4, -5, -6, -7, -8, -9, -10, -11, -12, -13, -14, -15, -16, -17, -18, -19, -20

initA: ; initialise the A pointer

ldi ZH, high(A\_P<<1)

ldi ZL, low(A\_P<<1)

ldi XH, high(A)

ldi XL, low(A)

clr r16

clr r17

loadA:

cpi r16, 20

breq initB

lpm r19, Z+

st X+, r19

inc r16

rjmp loadA

initB: ;initialise the B pointer

ldi ZH, high(B\_P<<1)

ldi ZL, low(B\_P<<1)

ldi YH, high(B)

ldi YL, low(B)

clr r16

loadB:

cpi r16, 20

breq initAdd

lpm r19, Z+

lpm r20, Z+

st Y+, r19

st Y+, r20

inc r16

rjmp loadB

initAdd: ;initialise the pointers

ldi ZH, high(C)

ldi ZL, low(C)

ldi YH, high(B)

ldi YL, low(B)

ldi XH, high(A)

ldi XL, low(A)

addArray:

cpi r17, 20

breq exit

ld r19, X+ ;load val of A

ld r20, Y+ ;load val of B

ld r21, Y+ ;load second half of B

ldi r18, 0

add r19, r20 ;adds val of A and B

adc r18, r21 ;adds second part of B

st Z+, r19 ;stores in C

st Z+, r18 ;C must be two bytes

inc r17 ;inc counter

rjmp addArray

exit:

nop

**//The code below is incomplete, leaving for existing comments.**

;assuming A and B are in the program memory and C is in the data

.dseg

C: .byte 40

.cseg

ldi YH, high(A<<1)

ldi YL, low(A<<1)

ldi XH, high(B<<1)

ldi XL, low(B<<1)

ldi ZH, high(C)

ldi ZH, low(C)

clr r17 ;counter

store:

cpi r17, 20

breq exit

ld r19, Y+ ;load val in A

ld r20, X+ ;load val in B

ld r21, X+ ; load second half of B

ldi r18, 0

add r19, r20 ;adds val of A and B

adc r18, r21 ;adds second part of B

st Z+, r19 ;stores in C

st Z+, r18 ;C must be two bytes

inc r17 ;inc counter

rjmp store

exit: nop

d) Store the string 12345678 into program memory using .db and .dw.

(v The extra zero is to make sure that the data is word aligned)

String: .db “12345678”, 0, 0

String2: .dw ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, 0, 0

e) Load the values stored in the program memory in (d) and store them into data memory in the reverse order.

.include "m2560def.inc"

.dseg

doge: .byte 8

.cseg

setup:

ldi r17, low(RAMEND)

out SPL, r17

ldi r17, high(RAMEND)

out SPH, r17

ldi zh, high(String<<1)

ldi zl, low(String<<1)

ldi yh, high(doge)

ldi yl, low(doge)

//to act as terminating byte for popping to work

pushZero:

ldi r17, 0

push r17

main:

lpm r17, z+

cpi r17, 0

breq reverse

push r17

jmp main

reverse:-

pop r17

st y+, r17

cpi r17, 0

breq end

jmp reverse

end: nop

**23. How do you multiply a two byte number by a one byte number? (Explain using a simple example). Do we have to consider the carry bit in the STATUS register for this case?**

[**https://sites.google.com/site/avrasmintro/home/2b-basic-math**](https://sites.google.com/site/avrasmintro/home/2b-basic-math)

^ This may helps?

Super simple example: multiplying 1x2 is easy -> it's 2. multiplying 98x76 is harder, because there's two digits. you need to carry across, eg

98

x 76

(6x8 + 6x90)

+ (7x8 + 7x90

= 7448

**Alternatively, you could split the multiplication up.**

3 \* 260 = 3(255 + 5) = 3\*255 + 3\*5

low ^ ^ high

ldi r16, 3

ldi r17, high(260) ;not sure about endianness

ldi r18, low(260)

clr r20

mul r16, r17 ;3\*5 stored in r21:r22)

mov r21, r0

mov r22, r1

mul r16, r18 ;3\*255 stored in r23:r24)

mov r23, r0

mov r24, r1

add r22, r24

adc r21, r23

adc r20, r21 ;Max 3 byte representation

;Result stored in r20:r21:r22

I wrote the best answer to this but it got deleted so whatever:

multiplying two byte by one byte is the same as above,

12 34

x 00 56

34 \* 56 +

12 \* 56 \* 256 [like multiplying by ten]

So you'll need to do it in several steps: multiply the least significant bytes, that's a two byte result [possibly with carry]

multiply the one byte with the MSB of the two byte adding onto the MSB if the carry register is set. and multiply that by 255.

then add them together

disclaimer: this could be wrong, let me know?

// Attempt at answer

Answer: r4:r2

Two byte number: r17:r16

One byte number: r18

MUL R16, R18

MOVW R3:R2, R1:R0

MUL R`17, R18

ADD R3, R0

ADC R4, R1 ; carry used

//What I think the full answer is.

Number Low = NL  
Number High = NH

Multiplier = M

Result goes back into NH and NL

mul NL, M

mov r2, r1

mov NL, r0

mul NH, M

mov NH, r0

add NH, r2

//End of my answer

This is my answer (in comic sans so take me super srs). Imagine a one digit decimal multiplication with a two digit, for example 4 times 25. 4 x 25 = 4x5 + 4x2x10.

Now imagine we are working in base 256 (i.e. one digit is one byte). The same method applies. Multiply the least significant byte, then add this to {the most significant byte x 256 multiplied with the one byte number}. Or symbolically:

[A|B] x [C] = CxB + AxCx256. The answer could be up to THREE bytes (although truncation can be done). How to implement this is where you get creative.

NOTE this is assuming unsigned multiplication.

**24. Investigate the different ways of writing AVR assembly code for the following scenarios**

// should we write more than one method for it?

// if your idea is better/easier to remember/shorter/etc, go for it (don’t delete the old one)

a) Copying a pair of registers into another pair of register.

Eg.

ldi r16, 1

ldi r17, 2

movw r21:r20, r17:r16

alternative (only need to supply the low bytes)

movw r20, r16

#### b) Multiply a number by 4.

1.Logical shift left twice.

lsl Rd

lsl Rd

2.Time: ldi Rd, (4\*Number)

3.Shift: ldi Rd, (Number<<2)

4. mul Rd, 4

mov Rd, r0

5. <<2

#### c) Divide a number by 4.

1.Logical shift right twice.

lsr Rd

lsr Rd

//V Probably not correct

2.Time: ldi Rd, (Number/4) ; Please check if im correct!!! Confirmed that this works in AVR, compiled and executed successfully.

3.Shift: ldi Rd, (Number>>2) ; Check if im correct please!!! Confirmed that this works in AVR, compiled and executed successfully.

4, >> 2

**25. When are MUL, MULS and MULSU instructions used and how are they are used?**

**Write AVR assembly code to perform multiplication for the following set of numbers,**

MUL, multiplication of unsigned integers.

MULS, multiplication of signed integers.

MULSU, multiplication of a signed integer with an unsigned integer.

a) 10, 12 (1 byte result)

ldi r16, 10

ldi r17, 12

mul r16, r17

mov r20, r0 ;optional (used to move the result)

b) -11,11 (1 byte result)

ldi r16, -11

ldi r17, 11

mulsu r16, r17

mov r21, r0 ;optional (used to move the result)

c) -4,-14 (1 byte result)

ldi r16, -4

ldi r17, -14

muls r16, r17

mov r22, r0 ;optional (used to move the result)

d) 32,258 (2 bytes result)

ldi r16, 32

ldi r17, low(258)

ldi r18, high(258)

mul r16, r17

mov r23, r0

mov r24, r1

mul r16, r18

adc r24, r0

e) -352, 28 (2 bytes result)

ldi r16, 28

ldi r17, low(-352)

ldi r18, high(-352)

mulsu r16, r17

mov r25, r0

mov r26, r1

mul r16, r18

adc r26, r0

// Can confirm this (v) works tested on AVR

ldi r16, 28

ldi r17, low(-352)

ldi r18, high(-352)

mul r16, r17

movw r21:r20,r1:r0

mulsu r16, r18

adc r21, r0

f) -27,-375 (2 byte result) //tested with AVR.

ldi r16, -27

ldi r17, low(-375)

ldi r18, high(-375)

muls r16, r17

mov r27, r0

mov r28, r1

mul r16, r18

adc r28, r0

// Can confirm this (v) works tested on AVR

.cseg

ldi r16, low(-375)

ldi r17, high(-375)

ldi r18, -27

mulsu r18, r16 ; signed being the -27 and unsigned being low(-375)

movw r21:r20, r1:r0

muls r18, r17

add r21, r0

// For e) Tried to modify using mul for low bytes and mulsu for high bytes it still gives the wrong result: 0x DA 80

clr r16

clr r17

clr r18

clr r25

clr r26

ldi r16, 28

ldi r17, low(-352)

ldi r18, high(-352)

mul r16, r17

mov r25, r0

mov r26, r1

mulsu r18, r16

adc r26, r0

end: rjmp end

**26. 1 Minimally modify the code below to add two numbers (in r17:r16 and r19:r18) when the result is bigger than 255.**

**ldi r16, 1**

**ldi r17, 0**

**ldi r18, 255**

**ldi r19, 0**

**add r16, r18**

**add r17, r19**

Change ‘add r17, r19’ -> ‘adc r17, r19’

**26.2 Write AVR code to add two 32 bits values?(Using R16-R23 to hold all values.)**

**a = 0x00000100**

**b = 0x002000FF**

ldi r16, 0x00

ldi r17, 0x01

ldi r18, 0x00

ldi r19, 0x00  
ldi r20, 0xFF

ldi r21, 0x00

ldi r22, 0x20

ldi r23, 0x00

add r16, r20

adc r17, r21

adc r18, r22

adc r19, r23

**27. Please complete the following table with instructions used for each operation.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Instructions** | **Registers** | **Stack** | **Data Memory** | **Program Memory** | **Separate IO** | **Memory-Mapped IO** |
| **Initialize** | clr/ser | ldi YH, high(RAMEND)  low  out SPH, YH | .dseg  .byte | .cseg  .db  .dw | **output:** ser r16  out DDRx, r16 | **input:**  clr r16  sts DDRx, r16 |
| **Write to** | ld, ldi, lds, ldd, mov | push | st/sts/std | spm | out | sts |
| **Read from** | mov | pop | ld/lds | lpm | in | lds |

**28. How do you setup a port to act as an input port or as an output port in AVR? What instructions are used to read from an I/O port? What instructions are used to write to an I/O port?**

Input = 0000 0000 (clr makes a register all 0s)

Output = 1111 1111 (ser makes a register all 1s)

DDRx = Data direction register (X ← whatever port e.g. A B C D etc)

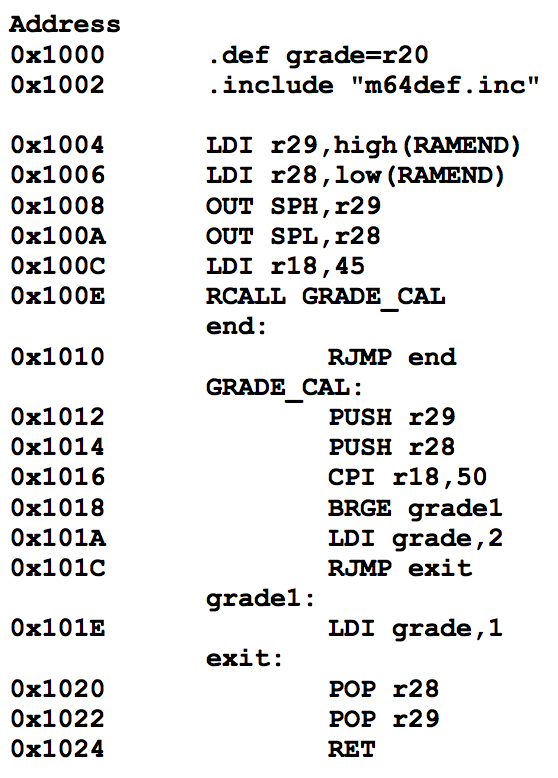
Input port: Output port:

clr temp ser temp

out DDRx, temp out DDRx, temp

IN/OUT are used to read to the first (64?) of I/O ports, and STS/LDS are used to read/write from the other ports.

**29. Consider the following example AVR code segment:**



What are the values of r28, r29, SPL and SPH:

From m2560def.inc:

.equ RAMEND = 0x21ff (end of internal SRAM)

.equ XRAMEND = 0xffff (end of external SRAM)

#pragma AVRPART MEMORY INT\_SRAM SIZE 8192

#pragma AVRPART MEMORY INT\_SRAM START\_ADDR 0x200

a) after line “LDI r28,low(RAMEND)”?

|  |  |  |  |
| --- | --- | --- | --- |
| r28 | r29 | SPL | SPH |
| 0xFF | 0x21 | 0x00 | 0x00 |

b) after line “OUT SPL,r28”?

|  |  |  |  |
| --- | --- | --- | --- |
| r28 | r29 | SPL | SPH |
| 0xFF | 0x21 | 0xFF | 0x21 |

c) after line “BRGE grade1”?

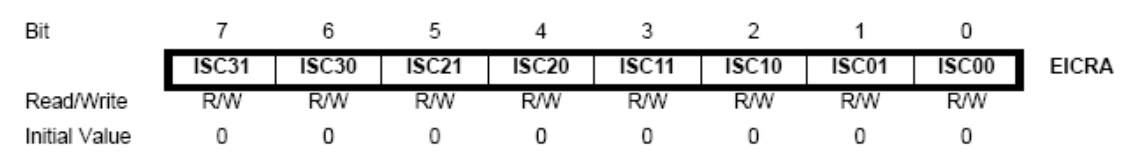
|  |  |  |  |
| --- | --- | --- | --- |
| r28 | r29 | SPL | SPH |
| 0xFF | 0x21 | 0xFA | 0x21 |

d) after line “POP r29”?

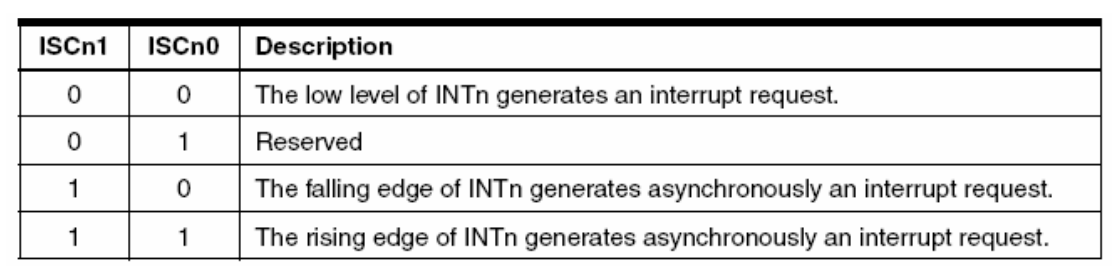
|  |  |  |  |
| --- | --- | --- | --- |
| r28 | r29 | SPL | SPH |
| 0xFF | 0x21 | 0xFC | 0x21 |

(3 bytes for rcall, 1 byte for each register)

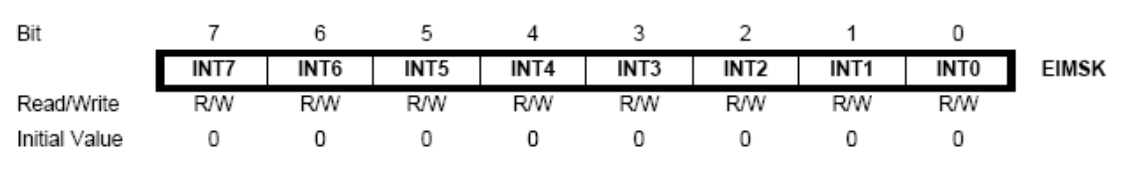
**30. The EICRA register is used to indicate what condition should be present for external interrupts to occur, and looks like this:**



**where each pair of bits ISCn1 and ISCn0 mean the following for INTn:**

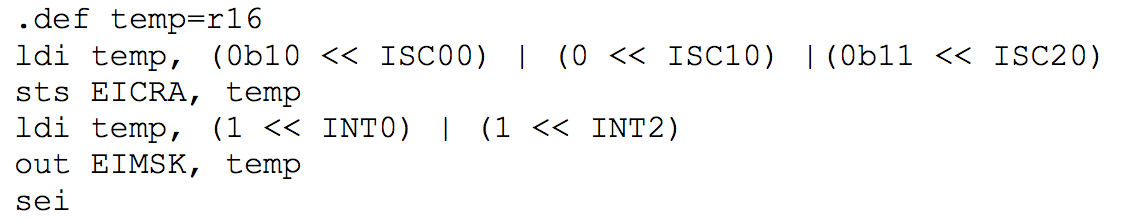


**The EIMSK register is used to enable the external interrupts and looks like this:**



**In “m64def.inc”, the values in these registers have been defined to their bit value. e.g., ISC00 = 0, ISC11=3 and INT2=2. Knowing this, examine the following code:**

a) What is the value (in binary) that is written to the EICRA register?

The instruction basically ORs together a bunch of values which are shifted left by various amounts (for example, ISC00 = 0, ISC11 = 3)  


|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ISC31 | ISC30 | ISC21 | ISC20 | ISC11 | ISC10 | ISC01 | ISC00 |
| 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 |

*Note: Unspecified values are equal to 0*

So, the value stored in temp, and thus written out to EICRA, is 00110010

b) Why do we use this approach to set up the register values?

To make it clear to somebody reading the code what values, and thus settings, we are assigning for each external interrupt

c) Which external interrupts can occur, and when will they occur?

External interrupt 0 can occur, triggered by a falling edge signal. External interrupt 2 can occur, triggered by the rising edge of a signal.

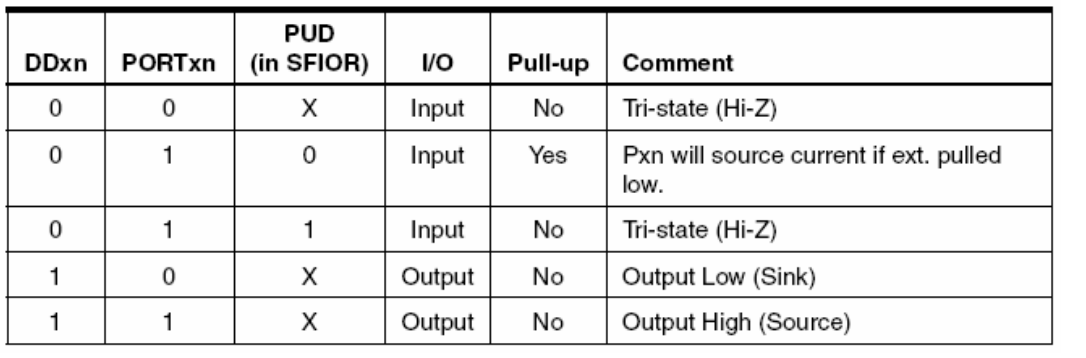
d) What is the difference between the ‘sts’ instruction and the ‘out’ instruction?

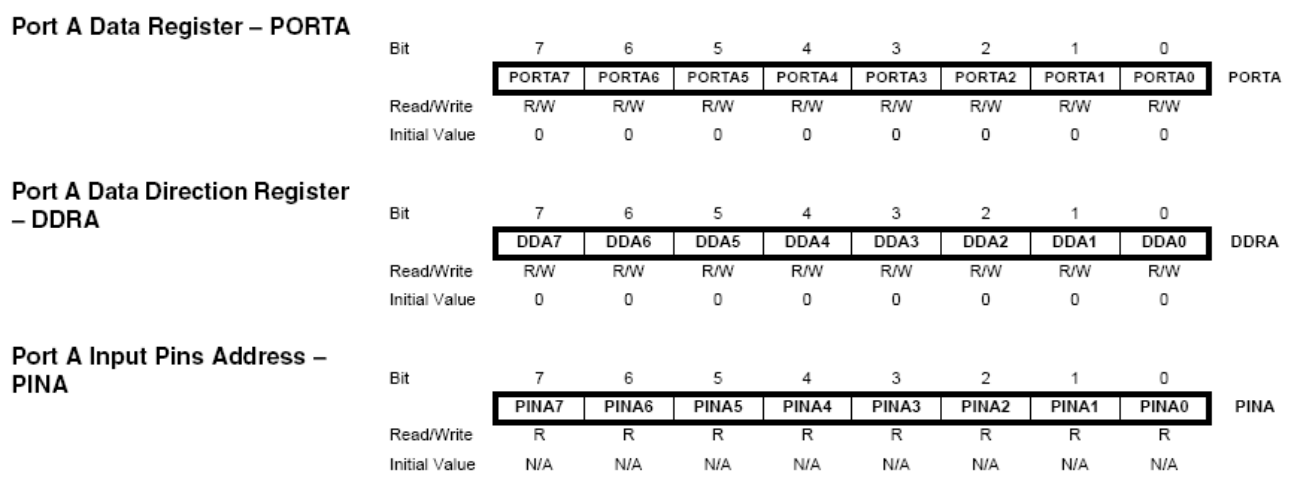
sts writes to a memory mapped I/O register, while out is used for I/O ports with designated registers?

STS has access to entire memory space? Yes

OUT has a limited range just inside I/O memory? Yes

**31. This question looks at the registers associated with PORT A. The following tables might help:**





a) What is the purpose of the DDRA register?

DDRA register stores I/O mode of port: 0 = input, 0xFF = output. Additionally, individual pins can be set to I/O by clearing or setting their individual bit, so for example, half a port could be used as input and half as output (eg, DDRA = 0x0F).

b) What is the purpose of PORTA0 when DDA0 = 1?

PIN0 on PORTA will be used as output.

c) What is the purpose of PORTA0 when DDA0 = 0 and PUD = 0? OI...you know what this shit means? ...PUD=0. LEL WTF?

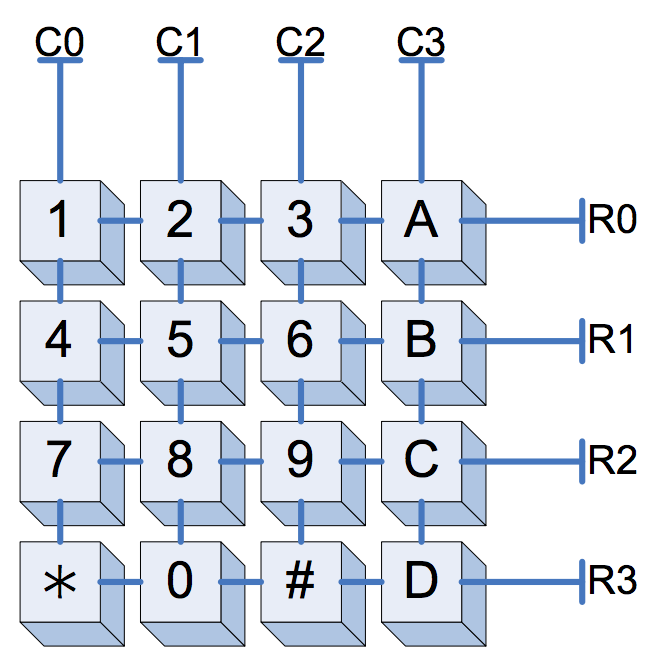
PIN0 on PORTA will be used as input. Setting bit PORTA0 will enable pull up resistors on PIN0. Clearing the bit mean PIN0 will be in high impedance mode (Hi-Z). Use pull up resistors when the connection on the pin will sometimes be floating, such as when connected to an open switch.

d) What is the purpose of the PINA register?

PINA stores values of PORTA pins (which pins are high, which pins are low).

PINA is used to read input from the port if port is set to input mode (DDRA = 0x00)

**32. The Keypad on the AVR boards is a set of 16 push buttons. The keypad has four rows and four columns, accessible via the pins R0-R3 and C0-C3. When you push a button on the keypad, it connects the column of the key to the row of the key as follows:**



One method to correctly read what keys are being pressed is to:

1: Set up the rows so that they read a Logic 1 when none of the buttons on the row is pushed.

2: Set one column to Logic 0 and all other columns to Logic 1.

3: Read the values of the row pins. If a row reads as Logic 0, you know that the switch at that row and column must be pushed.

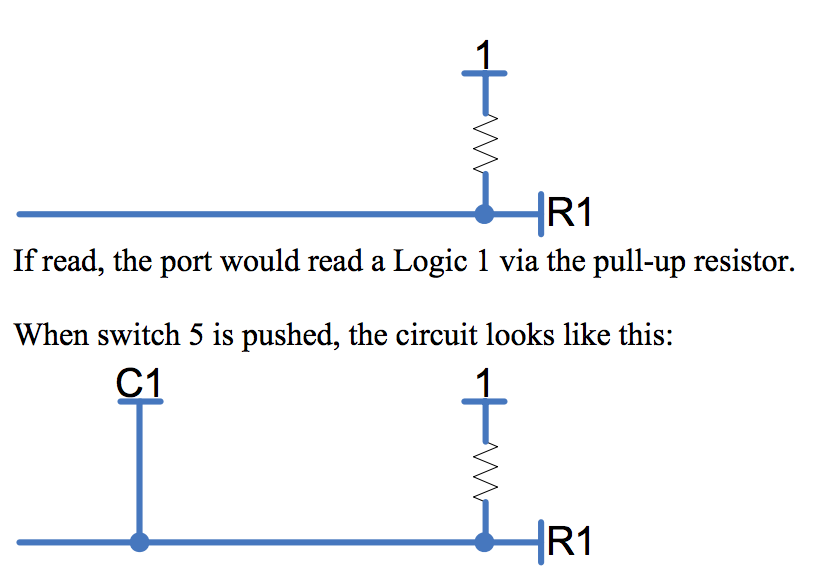
4: Set a different column to Logic 0 and read the rows.

5: Repeat steps 3 and 4 until a switch is found to be pressed or you run out of columns.

6: Repeat steps 2-5 again if you want to see whether a different switch is pushed.

Part of your third lab requires you to perform this algorithm. Steps 2-5 should be fairly simple to code, but step 1 is not so obvious. The way to accomplish this is with pull-up resistors. Pull-up resistor ties an input pin to Logic 1 via a resistor. This means that an input pin will still read any value that is input, and will read Logic 1 if disconnected.

To further understand this, look at switch 5 in the above diagram. When none of the switches connected to row 1 are pushed, the circuit (with pull-up shown) looks like this:



In this case, the port connected to R1 will always read the current value of C1. When C1 is Logic 0 there will be a voltage drop across the resistor, but this will not affect the value being read. Thus, the pull-up resistor accomplishes the desired task.

a) How do you setup an AVR I/O port so that it has pull-up resistors connected to its input pins? (See question 2 of this tute)

From the lecture notes(week 4, p43):

When the pin is configured as an input pin,the pull-up resistor can be activated/deactivated.

To active pull-up resistor for input pin, PORTxn needs to be written logic one.

So DDRx = 0 and PORTx = FF?

b) Write the code to find a switch that has been pushed by scanning either the columns or rows. (You have to do this for your lab, anyway)

c) Can you see an electrical problem with this scanning method when two switches on the same row are pushed at the same time (e.g., 5 and 6)? How could you correct this?

**(Hint: There might be something better you can do than output logic 1s to the columns you are not testing.)**

Two shorts are created. The ultimate solution is having a diode on all switches to only allow current to flow to the column outputs at the bottom of the circuit.

//please add more to answer guis

# Extra Qns (by Oliver Tan)

# Numbers Questions

<https://docs.google.com/document/d/1i-_rCVuaJfo4biF-G9PttJkhioAO8T9bUFw8kZbX32I/edit>

1. Which of the following rows have equivalent values:

|  |  |  |  |
| --- | --- | --- | --- |
| (a) | -7 | 000001112 | -0x7 |
| (b) | 256 | 0xF | 111111112 |
| (c) | 1000000002 | 256 | 0xF |
| **(d)** | **-2** | **0xFE** | **111111102** |
| (e) | 0x1 | 256 | 111111112 |

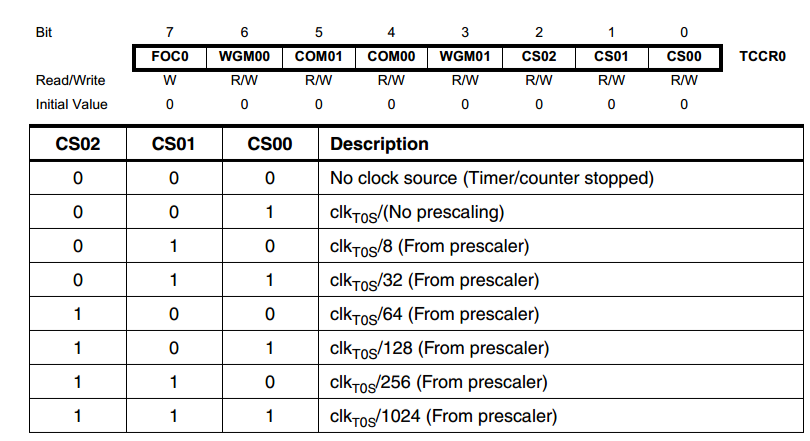
1. What is the binary result of 100111002 - 11110012 if it is an eight bit operation.
   1. 001000102
   2. **001000112**
   3. 010100102
   4. 010100112
   5. 1001000112
2. Which of the following statements is correct:
   1. When performing operations on four bit numbers, to store the result, I need at most 8 digits for addition and 8 digits for multiplication.
   2. When performing operations on four bit numbers, to store the result, I need at most 4 digits for addition and 4 digits for multiplication.
   3. **When performing operations on four bit numbers, to store the result, I need at most 5 digits for addition and 8 digits for multiplication.**
   4. When performing operations on four bit numbers, to store the result, I need at most 4 digits for addition and 8 digits for multiplication.
   5. When performing operations on four bit numbers, to store the result, I need at most 5 digits for addition and 16 digits for multiplication.
3. What is the hexadecimal representation of the result of adding 0x40D00000 and 0x40080000 if these numbers were of the IEEE 754 Single-Precision Floating Point standard:  
     
     
   Start with the hexadecimal values...  
   40D00000  
   40080000  
   Convert to binary and split into floating point formatting...  
   0 1000 0001 1010 000 0000 0000 0000 0000  
   0 1000 0000 0001 000 0000 0000 0000 0000  
   Convert to regular binary value and add  
    110.1  
   + 10.001  
   -------------  
   1000.101  
   Convert sum back to floating point formatting...  
   0 1000 0010 0001 0100 000 0000 0000 0000  
   Convert to hexadecimal value...  
   410A0000  
   1. 0x41C50000
   2. **0x410A0000**
   3. 0x41A50000
   4. 0x42528000
   5. 0x40940000

# Interrupt Questions

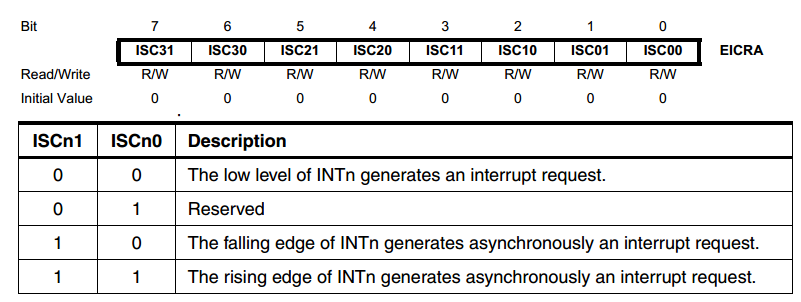
<https://docs.google.com/document/d/1ZFA1ddqZG8lig_eTLmQa0_Sd7V7OtoFDdP3aQxODTKs/edit>

* 1. Watchdog Timers
  2. Board Reset
  3. Keypad Pressing? (polling keypad instead of interrupts)
  4. CTRL+C on Ubuntu
  5. **Playing Music?**

1. Which of the following is NOT on the stack after an interrupt preamble (on the AVR)?
   1. a copy of the return address
   2. a copy of the status register(s)
   3. a copy of the conflict registers
   4. variables that are used in the previous function?
   5. **none of the above**
2. What does the interrupt do on the software side to activate the appropriate operations on the AVR board?
   1. The PC will change directly into the appropriate instruction, which is at a hardcoded location
   2. The program will automatically lookup “INT\_0\_HANDLER” in the code and the PC will jump to this location
   3. The PC will jump to the address indicated by the contents of 0x0002
   4. The PC will jump to the base of the function, which is determined by finding the number of iterations of “reti”
   5. **The PC will jump based on a interrupt vector table**



1. Given the image above, how would I load the temp variable so that when it outputs to TCCR0, the prescaler is set to 8?
   1. ldi temp, 010
   2. ori temp, (10 << CS01)
   3. ldi temp, 8
   4. ldi temp, (10 << CS00)
   5. **ldi temp, (2 << CS00)**



1. How would you load the “temp” register to, considering the interrupts which may have already been initialised, create an interrupt on the falling edge of INT3 when the temp register is loaded into EICRA?
   1. ldi temp, (1 << ISC30)
   2. ori temp, (1 << ISC30)
   3. ori temp, (10 << ISC30)
   4. ldi temp, (1 << ISC31)
   5. **ori temp, (1 << ISC31)**

# 

# 

# 

# SAMPLE FINAL EXAM 2

<http://www.cse.unsw.edu.au/~cs2121/LectureNotes/wk14.pdf>

## 1. Basics Concepts (12 × 3 = 36 marks)

### PART 1 Describe the difference(s) between microprocessors and microcontrollers. What is ATmega64?

A Microprocessor is an integrated circuit which only contains a CPU (such as intel’s pentium processors). Microcontrollers has a CPU, RAM and other peripherals embedded on a single chip.

ATMega64 is an 8-bit RISC microcontroller made by Atmel. It features 32 8-bit registers, pipelining, 64 I/O registers and is based on the modified Harvard Architecture.

### PART 2 Explain the concept of endianness. Which endianness does AVR use? Please give an example.

Endianness describes the order in which bytes are stored in memory. Big Endian refers to the order where the most significant byte is stored in the lowest memory. Little Endian refers to the order where the least significant byte is stored in the lowest memory.

0x123456

|  |  |  |
| --- | --- | --- |
| Address | Little Endian | Big Endian |
| 1000 | 56 | 12 |
| 1001 | 34 | 34 |
| 1002 | 12 | 56 |

AVR uses little endian.

### PART 3 Function can be recursive. Can macro be recursive? Why?

No they can’t. Macros are pre-processed unlike how recursive functions work during runtime, and would output an error. Limited nested Macros are possible though.

(^Not too sure about this)

### PART 4 Describe differences between memory-mapped I/O and separate I/O.

Memory mapped i/o means that all io is mapped to the same set of addresses as the rest of memory. Separate io is would be on a different set of addresses. The end result would just be a different instruction would be needed to do io whereas the same can be used as for memory w/ memory mapped io.

AVR supports both memory-mapped I/O and separate (i.e. port-mapped) I/O.

### PART 5 In ADC, what is resolution? And what is accuracy? What are the differences between these two terms? Please give an example.

Resolution is the number of output levels produced as a result of quanween the quantised output with the original value.tisation, whereas accuracy is the difference bet

// my answer

Resolution - smallest analog value that produces a digital code

Accuracy 1 100% \* (V\_resolution / V\_signal)`010100 = ??

### PART 6 What is stack frame? Draw a memory map to show the basic structure of a stack frame. Please list instructions that can access stack in AVR.

Each function call creates a new stack frame on the stack. The stack frame contains a return address, the conflict registers and local variables.

Push and pop can access the stack in AVR.

### PART 7 What is watchdog timer? What should be done to set up watchdog timer before to use it?

Watchdog timer is a timer that overflows at regular intervals. The running program detects the overflows of the watchdog timer, and if the timer “skips” an overflow, an error has occurred in the program. Watchdog timer is basically a way to ensure that program is running normally.

Prior to using the watchdog timer, WDE and WDCE bits in the WDTCSR must be set.

### PART 8 What is aliasing in ADC? How to avoid aliasing?

An undersampled signal, when converted back to into a continuous time signal, will exhibit ‘aliasing’. Aliasing refers to unwanted components being present in a reconstructed signal.

To avoid aliasing, sample at frequency F, where F is greater than or equal to the Nyquist rate (the Nyquist rate refers to sampling at twice the maximum frequency of the original signal). We know we need to sample at >= double frequency because of the Nyquist-Shannon sampling theorem. ***<- biggest bullshit ive read in a while. why tf do we need to know this lel...***

OR fmax = 1 / (2 \* convertion\_time).

### PART 9 What does USART stand for? In asynchronous serial communication, how is receiver clock is synchronized to a transmission operation of the transmitter?

Universal **Synchronous** Asynchronous Receiver Transmitter. In the frame, when logic one changes to logic zero, (mark goes down to space), the receiver detects the start bit, and is synchronised to the transmitter, from here it knows to start clocking in serial bits.

### PART 10 What is interrupt vector table? How many interrupts are available in Atmega64? The following is the part of interrupt vector table in an AVR program. Is this part of vector table correctly set? Why?

### ; interrupt vector table

### .org 0x0000

### rjmp RESET

### rjmp INT0

### rjmp INT1

**-- Assuming that we’re using a 2560, the ORDERING of the interrupts is correct: (INT2 directly follows INT1 which directly follows RESET). However, the rjmp instruction is only 1 word, which means what theyve written above is the equivalent of:**

**.org RESET**

**rjmp RESET**

**rjmp INT0**

**.org INT0addr**

**rjmp INT1**

**Solutions: Replace rjmp with jmp (two words) OR nop after each rjmp OR use .org for each case.**

An interrupt vector table consists of interrupt vectors. Each vector is the memory location of an interrupt handler, which is associated with an interrupt request.

Each interrupt vector is 4 bytes (i.e. 2 words) and typically contains a jmp to a subroutine to handle the interrupt because you cannot fit many instructions in 4 bytes. The RESET interrupt vector is located at address 0x0000 which means this ISR is the code executed upon a reset (as the program counter is set to 0, also pressing the reset button triggers an interrupt).

Fix it we add tags

.org 0x0

jmp RESET

.org INT0addr

jmp INT0

.org INT1addr

rjmp INT1

There are 8 interrupts available.

### PART 11 There is no software interrupt available in AVR. How can you implement a software interrupt in AVR?

From the page, it looks like the best way to do it is to pull int2 or something high **LOW/EDGE TRIGGED (no such thing as level high interrupts in AVR)**, causing the interrupt. It was also suggested to reset the interrupt register and jump to the interrupt function.

From my notes in the last lecture, Sri said, connect port to the interrupt pin and then output to the port. Not sure if this was exactly what he said though.

I would say (as per datasheet) - Activity in the EICRA or EICRB pins will trigger an interrupt request, even if the pin is set to output. This provides a simple way of generating a software interrupt.

### PART 12 The keypad is a typical input device in microcontroller application. Write a high level description (algorithm) that specifies how the input data from keypad is obtained by the microcontroller.

For each column from left to right

For each row from top to bottom

For each key being pressed

//For each key being scanned [I think it should be this, same as lecture notes]]

if key is pressed

display

wait

endif

endwhile

endwhile

repeat scanning process

// An alternative solution…? idk. lol

while (universe still exists) { // Worst case O(∞)

for (each column) {

for (each row) {

if (key is not pressed) continue;

wait 20ms; // wait for stability

if (key is not pressed) continue;

process keypress;

while (key is pressed) do nothing; // Wait for key release

}

}

}

## 2. Miscellaneous Questions (31 marks)

### PART 2 Consider the content of the AVR program memory in hex format as shown below. (a) What sequence of 4 ASCII characters does the content correspond to?

**0x3C = < , 0x4E = N , 0x53 = S , 0x5A = Z -** all of them?

### (b) What 2-byte signed integer values in base 10 does the content correspond to?

0x3C4E = 15438 (same if unsigned)

[isn’t program memory little endian - so 0x4E3C = 20038? if address is increasing left to right]

0x535A = 21338 (same if unsigned)

(c) What 2-byte unsigned integer values in base 10 does the content correspond to? (6 marks)



0x3C4E = 1543810

0x535A = 2133810

### PART 3 Consider the following AVR assembly code:

**.MACRO delay**

**loop: subi @0, 1**

**sbci @1, 0**

**nop**

**nop**

**nop**

**nop**

**brne loop ; taken branch takes two cycles.**

**.ENDMACRO**

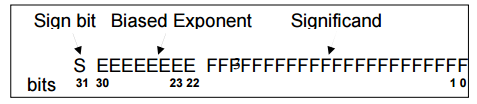
### All instructions in the program are 2 bytes long. What is the size of the code in bytes? How many parameters does the macro have? What is the range of values of each parameter? The code can generate a delay. What is the range of the delay? Assume the processor frequency is 8 Mhz. (7 marks)

14 bytes.

7\*2 = 14 bytes. Macro has 2 parameters. Byte ranges from 0 to 255, word (@1, @0) ranges from 0 to 65535.

1 microsecond to 65536 microseconds //This is 8 cycles

### PART 4 Consider two single precision floating point numbers x and y in IEEE 754 format, where x= 0x50240000 and y=0x40080000. The IEEE 754 format is given as follows:



### What is the decimal value of x+y? Please show your work. (4 marks)

PS: Sri Said in the revision lecture thingy not to account for precision loss. In any case, STATE YO ASSUMPTIONS.

0x50240000

0 101 0000 0 010 0100 0..0

sign: positive

exp: 160, so 160-127 = 33. so our num is \_\_ \* 2^33

mantissa:(implied 1) +0.25+0.03125 = 1.28125

so 1.28125 \* 2^33 = 11005853696

0x40080000

0 100 0000 0 000 1000 0..0

sign: positive

exp: 128, so 128-127 = 1. so our num is \_\_ \* 2^1

mantissa: (implied 1) +0.0625 = 1.0625

so 1.0625 \* 2^1 = 2.125

11005853696 + 2.125 = 11005853698.125

//this one sucks without a calculator - please add any other methods/tips/etc

I think i found a shortcut:

Continuing from above

0x50240000 -> 1.28125

0x40080000 -> 1.0625

1.0625 \* 2^1 < 1.28125 \* 2^33

We have to shift the 1.0625 by 2^32 times to get the same value exponents. BUT 32 > 23 bits (mantissa) so 1.0625 \* 2 -> 0.000 \* 2 ^ 33 (the result stored in the 23 bits will be 0) so we can add the values:

0.00 \* 2^33 + 1.28125 \* 2^33 = 1.28125 \* 2^33

Indeed if you plug the above tedius answer 11005853698.125 into a converter <http://www.h-schmidt.net/FloatConverter/IEEE754.html>

you get the same mantissa 1.28125 (also 11005853698.125/2^33 = 1.28125)

PS. I’m not very good at explaining, I got this trick from here: <https://www.youtube.com/watch?v=DuOwT2hZOiw>

Maybe someone can watch it and try to make it clearer?

Here’s my idea for smoothing things out with dealing with x.

I personally find it easier to go from:

1.01001\_2 \* (2^33)

Then doing a bit more pre-processing by multiplying it by a few multiples of 2 to get:

101001\_2 \* (2^28)

Which is:

41\*(2^28)

And should be easier to deal with by hand than decimal places.

### PART 5 How many bits do you need to represent a~z 26 letters and 0~9 ten digits? Can you encode them with the 4x4 keypad ? If no, why? If yes, how? (5 marks).

We will need to be able to represent 26+10=36 letters/digits. Lowest power of 2 is 64, so we will need a minimum of 6 bits to uniquely represent each letter and number.

*umm… help! can we encode them withld a 4x4 keypad?*

There are 36 symbols we need to encode.

Suppose each symbol was encoded by pressing two keys on the keypad

=> 16 choose 2 unique choices = 15\*16/2 = 120 > 36. So yes we can encode it on a keypad (given that ghosting is not an issue), though it wouldn’t be intuitive at all.

### 

## 3. AVR Assembly Programming and Design (33 marks)

### PART 1 Write an AVR assembly program to find the max value of an integer array, A. Your program must satisfy the following requirements.

### 1) Each element is a 2-byte signed integer.

### 2) Array A is stored contiguously in the FLASH.

### 3) Your program must define and use at least one MACRO.

### 4) The array length is 10. (7 marks)

//tested on avr studio after i finished this question with the two arrays i listed

//not sure if it works so well with signed numbers though, pls double check

.include "m2560def.inc"

.cseg

A: .dw 2000, 1, -200, 3, 0, 7777, 2, 4, 5, 6, 7

//A: .dw -500, -2, -50, -4, -10, -300, -7, -8, -9, -11

.macro StoreStuffYo

mov @0, @2

mov @1, @3

.endmacro

setup:

clr r16 //low byte of largest number

clr r17 //high byte of largest number

clr r20 //counter

clr r18 //low temp

clr r19 //high temp

ldi zh, high(A<<1)

ldi zl, low(A<<1)

//this dummy section was needed to initialise r16, r17, i was lazy so did it the easy way l0l

lpm r18, z+

lpm r19, z+

StoreStuffYo r16, r17, r18, r19

ldi zh, high(A<<1)

ldi zl, low(A<<1)

loop:

lpm r18, z+

lpm r19, z+

cp r18, r16 //compare temp variable with our actual return result

cpc r19, r17

brlt incCounter //if our temp variable is smaller, no point storing right?? so we continue

store: //this is where we move our temp variable into our return result

StoreStuffYo r16, r17, r18, r19

incCounter:

inc r20

cpi r20, 10

brne loop

end: rjmp end //rjmp is 2 cycles

**PART 2 An array of ten 2-byte integers are stored in the AVR program memory. Write a program to convert the array to different endianness format and store it back to the data memory. 5 Your program must satisfy the following requirements:**

### 1. Your program must use at least one function.

### 2. All local variables and parameters must be stored in the stack space.

### 3. You must describe the stack frame structures for the function used in your program. (8 marks)

### PART 3 Consider to design an embedded system to control the speed of a DC motor. The operating specification of the system is given below:

### 1. The speed of the motor is input from the keypad

**i. Assume that the motor is driven by a PWM signal. And there is a formula to determine the duty cycle for different motor speed when the motor spins without any extra load.**

### 2. The motor is started by an external push button operation

### 3. The speed deviation must be controlled within a specified limit.

### 4. If the speed deviation exceeds the limit, the LED bar is set to ON within 1 second to alarm the user and at the same time the motor is stopped.

### Your design should include:

### 1. definition of all tasks in your system.

### 2. task scheduling diagram that specifies the execution timing frame for each task

### 3. interrupt design that includes what kind of interrupts you are going to use and the purpose for using each of the interrupts

### 4. a code template that includes interrupt handling subroutines

### Note, the code template does not have to be complete. You can use comment lines for sections of code that involve detail setting information and can be inserted later. Examples are given below.

### ; insert code here to set up timer0 for 1 second timer out

### ; insert code here to enable timer0 overflow interrupt

### (18 marks)