CSC 230
Summer 2022 (A01: CRN 30207; A02: CRN 30208)
Midterm #2: Thursday, 7 July 2022

Marking Key

Students must check the number of pages in this examination paper before beginning to write, and report any discrepancy immediately.

- **All answers are to be written on this exam paper.**
- The exam is closed book. Other than the AVR reference booklet provided to you, no books or notes are permitted.
- When answering questions, please do not detach any exam pages!
- ***A basic calculator is permitted***. Cellphones must be turned off.
- The total marks for this exam is 70.
- There are ten (10) printed pages in this document, including this cover page.
- We strongly recommend you read the entire exam through from beginning to end before starting on your answers.
- **Please have your UVic ID card available for inspection by an exam invigilator.**

Question 1 (20 marks)

On this page are 20 terms, numbered from 1 to 20.

On the opposite page are 25 definitions, lettered from A to Y.

For each term on this page (page 2) choose the correct / best definition by writing the definition's letter beside the term.

Term	Definition (letter): *your answers*
1. local variables	W
2. hexadecimal	O
3. CPU	J
4. data memory	T
5. word	G
6. endianness	R
7. epilog	U
8. immediate	V
9. null-terminated string	D
10. bus	Q
11. stack pointer	P
12. computer system	X
13. pass-by-reference	K
14. directive	N
15. callee-save register	C
16. assembler	F
17. most-significant bit	H
18. call	M
19. bitwise AND	Y
20. run-time stack	I

A	An example of this in the assembler would be "`.direct`"
B	Needed for the AVR `onstack` and `offstack` opcodes.
C	It is **not** the responsibility of the code calling a procedure to save and restore the state of these variables.
D	A particular interpretation of character values in bytes of memory where the value of zero has a special meaning.
E	Core Programming Utility.
F	Makes multiple passes over a AVR program as it resolves labels into addresses.
G	In the AVR architecture, contents of a word would require two general-purpose registers to hold this value/quantity.
H	Left-most bit in a word.
I	Normally grows in the direction of lower memory addresses.
J	contains registers, control logic, program counter, ALU, etc.
K	A better choice for procedure arguments where the value of particular data argument may not fit within a single register.
L	A collection of instructions meant to be executed in a cycle bus.

M	With this instruction, a AVR processor modifies the instruction pointer.
N	An example of this in the assembler would be `.byte`
O	One of these digits represents four bits of information.
P	May be used within a procedure/subroutine to access parameters made available by the caller of that procedure/subroutine.
Q	One of these is for addresses.
R	This says something about the byte-order in memory for the storage of integers.
S	Links an AVR operation to the `listen` system call.
T	Region of memory that includes the run-time stack.
U	After this code is complete, the CPU may safely execute the `ret` instruction.
V	Normally represented in a AVR instruction using hexadecimal or decimal.
W	May be stored in a contiguous region of the stack known as a stack frame.
X	Depends upon buses to connect its different parts together.
Y	Can be used to implement a form of set intersection.

Question 2 (20 marks): Problem solving in assembly

Write in AVR assembly the function/procedure **`count_set_bits`**.

It uses one parameter pushed on the stack by the caller, and this parameter corresponds to an 8-bit unsigned integer. The function/procedure must determine the total number of set bits (i.e., bits set to 1) in the parameter and store the result in **`r17`** (i.e. this is how values are to be returned from this function/procedure).

For example, consider the following possible call to **`count_set_bits`**:

```
ldi r16, 0xD3
push r16
rcall count_set_bits
pop r16
```

The value expected here in **`r17`** is 5.

You are strongly encouraged to draw a stack frame as part of the explanation of your answer – with this, the marker can give partial marks if you make a mistake in some detail within your code answer. *Some marks will be given for the quality of your answer.*

You may use the next page for your answer.

The contents of the stack frame will look something like this:

0xD3
ret ??
ret ??
ret ??

The top three values on the stack (denoted "ret ??") will be the word address corresponding to the pop r16 instruction. Below these three bytes is the value passed as a parameter. Therefore to obtain that parameter, we can either pop off values (and re-push them), or we can use the X, Y, or Z register to hold the value of SP, and then add 4 to this value to obtain the memory address of the parameter.

(You may use this page for your answer to Question 2.)

```
; Note that we accepted as valid any answers which did *not*
; save and restore values. The main ideas probed in this question
; was the use of the stack for parameter values, and the ability
; to write a simple loop for a straight-forward problem.

count_set_bits:
    in YL, SPL
    in YH, SPH

    ; r17 will hold the value for which bits are counted
    ; r18 will hold the count of set bits
    ; r19 is a scratch register (holding bitwise-AND results)
    ldd r17, Y+4
    clr r18

count_set_bits_loop_top:
    mov r19, r17
    andi r19, 0x01
    breq count_set_bits_shift_right
    inc r18
count_set_bits_shift_right:
    lsr r17
    brne count_set_bits_loop_top
count_set_bits_exit:
    mov r17, r18
    ret
```

- **Correctly accessing parameter: 5**
- **Counting bits in a loop: 12**
- **Returning value in r17: 3**

Question 3 (20 marks): Short answers

a) What register is modified by mega2560's **`rjmp`** instruction? Explain. *[4 marks]*

The only register modified is the program counter or PC. The PC value is changed by adding to the PC the parameter given to rjmp plus 1.

b) If a **`breq`** instruction is immediately followed by a **`brne`** instruction, is it possible that neither branch will be taken? Explain. *[4 marks]*

Given that the breq instruction does not modify the status register – and hence will not modify the Z flag – then one of the branches must be taken. If the Z flag is set, the breq target will become the new PC value. If the Z flag is not set, then the brne target will become the new PC value.

c) What difficulty would be encountered if you wrote a function to add two bytes passed on the stack using the strategy of simply popping parameters off the stack and then pushing their sum? *[4 marks]*

The main difficulty is that we cannot immediately pop the parameter off the stack. That is, the return address sits between the top of the stack and the parameters. We can address the difficulty either by popping off the top three values and saving them in registers, then obtaining the parameters, followed by pushing the three values back onto the stack; or we can use relative address (via the ldd instruction) to directly add the memory locations in the stack at which the parameters are located. (For an example, see the solution to question 2.)

d) How does **`reti`** differ from **`ret`**? Explain. *[4 marks]*

reti is used for returning from an interrupt. ret is used for returning from a procedure / subroutine / function call. It would be a mistake to use reti when ret is expected, or vice versa.

(Note that there is one more difference between reti and ret – which is that the former sets the global interrupt flag in the status register – but this detail a little subtle and is not needed for full marks.)

e) In the context of an AVR timer, what purpose is served by the *prescaler*? *[4 marks]*

The prescaler acts as a clock divider – that is, the clock rate provided by default by the AVR mega2560 is "divided down" by the value of the prescaler. For example, if the clocks is running at 16 MHz (16,000,000 Hz), and the prescaler is set to 1024, then the timer for which the prescaler is applied will have its value incremented a rate of 16,000,000 / 1024.

Question 4 (10 marks): Stack

Write instructions to copy the stack pointer into **Y** and then use **ldd**, **inc**, and **std** instructions to increment the 8-bit number on the top of the stack.

Remember that **sph:spl** points to the byte just past the top of the stack (at the next lower address). This is why **ldd** is needed instead of **ld**.

Do not use **push** or **pop** in your answer.

The explanation of your solution may be in the form of comments with each of the lines. *Some marks will be given for the quality of your answer.*

```
in YL, SPL    ; need to fetch the current stack-top address
in YH, SPH    ; need to fetch the current stack-top address

ldd r16, Y+1  ; use direct addressing to access the value at
              ; the top of the stack

inc r16       ; add one to the value we just read from the stack

std Y+1, r16  ; ... and now write that value back
```

- **Ensure Y (or X or Z) is assigned SP: 4 marks**
- **Read the correct value from within the stack: 4 marks**
- **Add and store the computed value: 2 marks**

(Note: When evaluating the last bullet point, we have attempted not to double-dock marks – that is, mistakes already in the answer involving reading the stack should already be penalized earlier in this question.)

(You may use this page for your answer to Question 4.)

(This page intentionally left blank.)