CSC 230
Fall 2022 (A01: CRN 10817)
Midterm #2: Monday, 7 November 2022

Marking guide

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. callee-save register	V
2. null-terminated string	T
3. data memory	B
4. buses	S
5. epilog	C
6. run-time stack	F
7. assembler	D
8. Moore's law	H
9. directive	Q
10. stack pointer	A
11. most-significant bit	P
12. word	I
13. bitwise AND	K
14. pass-by-reference	Y
15. endianness	M
16. call	E
17. hexadecimal	X
18. immediate	W
19. CPU	N
20. interrupt	G

A	Value decreases as items are pushed, value increases as values are popped.
B	Region of memory that includes the run-time stack.
C	Code at the end of a procedure normally used to shrink the stack.
D	Converts AVR mnemonics into their encoded form.
E	With this instruction, the AVR processor modifies the PC register.
F	Suitable for, amongst other things, parameters and local variables/state in a function/procedure
G	Necessary to enable some I/O devices to interact with the CPU.
H	Has led to a form of exponential growth in transistor density over the past 40 years.
I	In the AVR architecture, consists of 16 bits.
J	A particular CPU instruction that a programmer uses to pause execution at some location.
K	Used as part of some bit-masking operations
L	The system of equations that determines a CPU's instruction set.

M	This says something about the byte-order in memory for the storage of integers.
N	Consists of sequential-logic and combinational-logic components
O	The equivalent operation for addition (as in `r3 AND r4 AND r5`).
P	Left-most bit in a word
Q	An example of this would be `.byte`.
R	Can be distinct from values kept in callee-save registers in that their values are read to and written from stack.
S	One of these is for data.
T	A particular interpretation of character values in bytes of memory where the value of zero has a special meaning.
U	One of the `cX` registers (e.g., `c0`, `c1`, etc.)
V	Value should be the same before and after a procedure call, assuming procedure is correctly written.
W	A literal value provided to some AVR instructions.
X	Another name for base 16.
Y	A style of parameter passing where the intention is to give the address of a value for the procedure.

Question 2 (four marks): Short answer

What registers are modified by the AVR **`ret`** instruction? Explain.

Since ret is the last instruction executed in a function, **the PC register is modified** control flow is meant return to the instruction following the original call.

And as that address is stored on the stack, and because ret causes that return address to be popped off the stack, **the stack register is also modified**.

Question 3 (four marks): Short answer

If a **`brne`** instruction is immediately followed by a **`breq`** instruction, is it possible that neither branch will be taken? Explain.

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.

Question 4 (four marks): Short answer

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 back onto the stack?

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.

Question 5 (four marks): Short answer

How does **`adiw`** differ from **`add`**? Explain.

The main difference is that **`adiw`** is used to **`add`** an immediate value to a pair of registers used as a memory address. The **`add`** instruction is meant to add the values of two separate registers together. For example, **`adiw Z, 4`** will add four to the register pair **`r31:30`**, but **`add r31, r30`** adds the contents of **`r30`** and **`r31`** together while storing the result in **`r31`**.

Question 6 (four marks): Short answer

Explain what the `ADC` is in the mega2560, and how it is read.

This is the analog-to-digital convert, and converts an analog voltage into a value from 0 to 1023. Its value is read by two separate operations (to read the low byte and the high byte), but only once the value has been requested and is ready (which usually requires a polling loop of some kind).

BUT ALSO...

Some interpreted ADC to mean the Add With Carry operation, and so interpreted the question in that spirit. Marks were given for this answer, although any future questions on tests regarding the analog-to-digital converter will explicitly mention it by name.

Question 7 (10 marks): Functions

Write in AVR assembly a function named **`look_at_either_end`** which takes a parameter passed-by-value in `r16`.

- If bit 0 is set in the parameter, but not bit 7, then return the value 3.
- If bit 7 is set in the parameter, but not bit 0, then return the value 1.
- If both bits 0 and 7 are set in the parameter, then return the value 2.
- For any other combination of bits in the parameter, return the value 0.

The return value is to be placed `r25`.

Note that there are ~~fetwow~~ a few other restrictions:

- You may not destroy the value passed into `r16` (or rather, the value in `r16` when the function starts must be the same value in `r16` when the function returns).
- In your solution you must not use `sbr`, `sbrc`, or `sbrs`.

Some marks will be given for the quality of your answer.

There are clearly many approaches towards a solution. However, the central idea will have to be some form of checking for each of the four possibilities (or rather, three possibilities plus a default).

Solutions need not be perfect. Although students had the instruction reference with them, there might be some infelicities such as missing semicolons or inconsistent labels.

(You may use this page for your answer to Question 7.)

One possible solution

```
look_at_either_end:
            push r17

            clr r25

            mov r17, r16
            andi r17, 0b10000001    ; 0x8a -- mask out bits 7 and 0

            cpi r17, 0b00000001     ; bit 0 set, but not bit 7?
            breq return_value_3

            cpi r17, 0b10000000     ; bit 7 set (but not bit 0)?
            breq return_value_1

            cpi r17, 0b10000001     ; both bits 7 and 0 set?
            breq return_value_2

            ; must be any other combination then.
            rjmp look_at_either_end_exit

return_value_3:
            inc r25

return_value_2:
            inc r25

return_value_1:
            inc r25

look_at_either_end_exit:
            pop r17
            ret
```

Question 8 (20 marks): Functions plus stack

Write in AVR assembly the function named **`loop_based_multiply`**.

It uses two parameters pushed on the stack by the caller, and these correspond to two 8-bit unsigned integers.

- You may assume that the stack was properly initialized elsewhere (i.e. earlier) in the program.
- You may assume that a previous assembler directive has include `m2560def.inc` for you to use.
- The values pushed onto the stack will always be positive numbers (i.e. never zero).
- You are not permitted to use any of the AVR multiply instructions in your answer (`MUL`, `MULS`, `MULSU`).

The function/procedure is to compute the product of the two parameters pushed onto the stack, with the result stored in `r25:r24`. The way this is to be done within your solution is by repeated additions via a loop.

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

```
ldi r16, 25
push r16
ldi r16, 20
push r16    ; this was missing in the original question
rcall loop_based_multiply
pop r16
pop r16
```

The value expected here in `r25:r24` is `500` (or `0x01F4`). (You can choose to have 25 added to itself 20 times, or 20 added to itself 25 times.) *Your solution must, of course, work for other parameter values than in the example.*

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.

(You may use this page for your answer to Question 8.)

There are clearly many approaches towards a solution. However, the central idea will have to be some loop that counts up or counts down, along with the use of add and adc in order to correctly add 16-bit values.

Solutions need not be perfect. Although students had the instruction reference with them, there might be some infelicities such as missing semicolons or inconsistent labels. *Answers may or may not have been written in a way to deal with the missing push.*

- **20 marks: Solution looks plausible, clearly explained**
- **16 marks: Solution looks plausible although there appears to be a single clear logic error or missing action (e.g. saving and restoring register used; result not placed into correct registers; etc. etc.)**
- **12 marks: Solution looks plausible although there appears to be two or three logic errors, or the actual work of adding two 16-bit integers is missing / not sensible (e.g. saving and restoring register used; result not placed into correct registers; etc. etc.)**
- **10 marks: Serious attempt at a solution, could be made to work with effort**
- **< 10 marks: Marks given for an attempt, although much more work is needed.**

```
; one possible solution
loop_based_multiply:
      push r31
      push r30
      push r16
      push r17
      push r18

      in ZL, SPL
      in ZH, SPH

      ldd r18, Z+9
      ldd r16, Z+10
      clr r17

      ; next column...
```

```
      clr r25
      clr r24

loop_based_multiply_loop:
      add r24, r16
      adc r25, r17
      dec r18
      brne
loop_based_multiply_loop

      pop r18
      pop r17
      pop r16
      pop r30
      pop r31
      ret
```

(This page intentionally left blank.)