TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

ACE 411 – EMBEDDED SYSTEMS

FALL SEMESTER 2021-22

Project report

E-class team: 2

Name: Alexandros Poupakis
A.M.: [redacted]

Name: Georgios Agoritsis
A.M.: [redacted]

Table of Contents

Project description

The goal of this project is to implement a 9x9 Sudoku solver on an AVR microcontroller. The microcontroller communicates with a controlling device, in this case a PC, via USART, according to the given communication protocol. Through this protocol, the microcontroller is supplied with the starting Sudoku grid, accepts various control commands and transmits the solved Sudoku to the controlling device.

The target device is the ATmega16, which offers 16KB programmable FLASH and 1KB internal SRAM. The chip is mounted on an STK-500 development platform, which provides connectivity to a PC both for programming and runtime communication, 8 built-in LEDs and wiring flexibility through its headers and jumpers. To develop our solution, we will be using the Microchip Studio IDE (version 7.0.2542) and the supported C and AVR Assembly programming languages.

The solution must meet the project's requirements, which can be split into two categories: functional and technical. The first functional requirement is the support of the given USART protocol. Secondly, the solver must find one valid solution to any – uniquely or otherwise – solvable Sudoku puzzle. Thirdly, the microcontroller must display the solver's momentary progress using 8 LEDs as a progress bar, where the number of lit LEDs corresponds to the number of decades of filled Sudoku cells. As for the technical requirements, the chip must be clocked at 10MHz, USART must run at 9600 baud rate and the progress bar should be updated with a frequency of 30Hz.

The evaluation process of the project is based on code structure, quality of documentation and, to a lesser extent, performance. Performance evaluation will measure the speed with which the implementation will find a solution to a given puzzle, ignoring the time delays of the communication interface.

USART Protocol

The valid commands of the given USART protocol are shown in the Tables below. The first table presents the available commands sent from the PC to the AVR, while the second table presents the available commands sent from the AVR to the PC.

Table 1: USART protocol commands from PC to AVR

User/PC command	Command Action	AVR response
AT<CR><LF>	–	OK<CR><LF>
C<CR><LF>	Clear Sudoku data (cell initialization) and clear (turn off) the progress bar.	OK<CR><LF>
N<X><Y><VALUE><CR><LF>	Insert cell value. <X>, <Y>, <VALUE>: ASCII characters, each corresponding to a decimal digit [1-9].	OK<CR><LF>
P<CR><LF>	Play – Start solving the Sudoku (no more input data are accepted).	OK<CR><LF>

S<CR><LF>	Send the value of the solved Sudoku's first cell.	N<0><0><VALUE><CR><LF>
T<CR><LF>	Send the value of the solved Sudoku's next cell (and update cell counter).	N<X><Y><VALUE><CR><LF> or D<CR><LF>
B<CR><LF>	Break – Stop the solving process or the result (cell values) transmission, without clearing and initializing the Sudoku data space.	OK<CR><LF>
D<X><Y><CR><LF>	Debug – Get the currently stored value of the cell (X,Y) from the AVR.	N<X><Y><VALUE><CR><LF>

Table 2: USART protocol commands from AVR to PC

AVR command	**Command Action**	**PC response**
D<CR><LF>	When the game solving process is completed, the AVR will send a DONE response to state that it is ready to start sending the results.	–

The protocol assumes error-free communication, transmission of valid commands only and therefore no need for validity checks. The USART is required to operate in asynchronous mode, with 8-bit frames, no parity, 1 stop bit and in full-duplex mode. The baud rate should be set to 9600.

For those settings, the USART Baut Rate Registers (UBRR) value is given by:

$$UBRR = \left(\frac{f_{osc}}{16 \cdot BAUD}\right) - 1 = \left(\frac{10MHz}{16 \cdot 9600}\right) - 1 = 64$$

where $f_{osc} = 10MHz$ is the required processor clock.

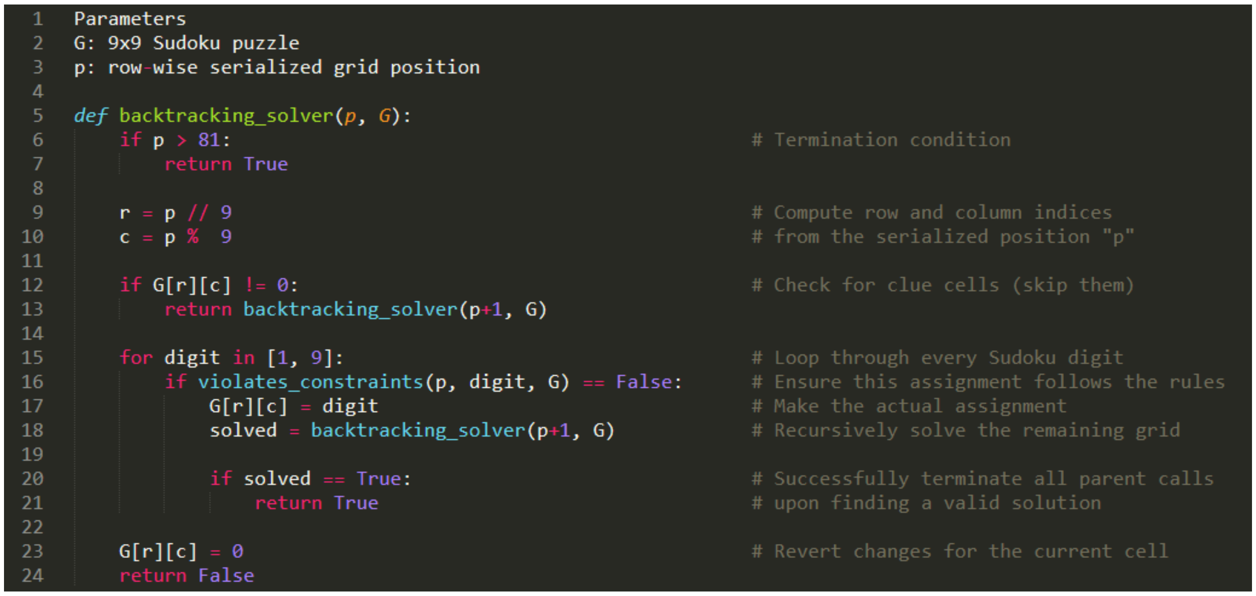
A well-known issue with USART communication is the unreliability of its cables. For this reason and for completeness purposes, we decided to remove the error-free communication assumption and set up a reliable and robust communication implementation, which assumes neither valid commands nor error-free transmission. For a USART protocol command to be valid, the whole ASCII character sequence must be successfully received by the AVR, in the correct order. In any other case, the received characters are automatically discarded up to and including the next received *<LF>* character.

Moreover, the protocol does not specify whether all commands are valid at any time/program state. For example, receiving a *break* command before a *play* command has been issued, or after a *done* response has been sent should not cause any action, yet the question of what the response should be remains open. Therefore, we define two program states, IDLE and SOLVING, each of which considers a specific subset of the protocol as valid commands. In the IDLE state, all defined commands are valid except for *B<CR><LF>* and *D<CR><LF>*. In the SOLVING state, the valid commands are *AT<CR><LF>, OK<CR><LF>, B<CR><LF>, D<CR><LF>*. When a command is received which is not allowed by the current state, no action is performed, no response is sent and the command is ignored.

Notice that the *invalid* commands will be successfully received by the communication protocol and only their respective actions will be skipped, during the action execution phase. The USART logic is shorter, simpler and therefore faster, by avoiding command validity checks based on the state of the program, during the character reception and command construction phase of the protocol.

<u>Backtracking algorithm analysis</u>

The simplest algorithm to solve Sudoku puzzles is the brute force approach with backtracking. This algorithm works by recursively visiting the Sudoku cells in some predefined order, assigning some digit to each of them until the Sudoku constraints are violated. At that point, the algorithm backtracks to the most recent cell for which at least one digit remains uncheck and sets the cell's value to that digit. The algorithm continues to visit the next cells again and repeats this process until either the entire grid is filled without violating the constraints, or all possible assignments have been checked but no solution exists. The algorithm's pseudocode (Python-style) is presented below.



```
Parameters
G: 9x9 Sudoku puzzle
p: row-wise serialized grid position

def backtracking_solver(p, G):
    if p > 81:                                         # Termination condition
        return True

    r = p // 9                                         # Compute row and column indices
    c = p %  9                                         # from the serialized position "p"

    if G[r][c] != 0:                                   # Check for clue cells (skip them)
        return backtracking_solver(p+1, G)

    for digit in [1, 9]:                               # Loop through every Sudoku digit
        if violates_constraints(p, digit, G) == False: # Ensure this assignment follows the rules
            G[r][c] = digit                            # Make the actual assignment
            solved = backtracking_solver(p+1, G)       # Recursively solve the remaining grid

            if solved == True:                         # Successfully terminate all parent calls
                return True                            # upon finding a valid solution

    G[r][c] = 0                                        # Revert changes for the current cell
    return False
```

Figure 1: Recursive backtracking algorithm

While the recursive method is exceedingly simple to write and understand, it is not at all optimized and comes with major speed penalties. Namely, recursion is expensive due to the function frames, the loop iterates through every digit even though some are guaranteed to be illegal, and the particular representation of the problem does not allow for fast constraint violation checks.

It is worth noting that the encoding and method with which the violation check is performed is neither specified, nor relevant. Clearly, a checking function that iterates through the entire grid would be tremendously slower than one utilizing some clever encoding and performing the check in a few cycles, but that is not the point. Regardless of optimizations in the checking function, the fact remains that a non-trivial – in terms of cycles – check must be performed for every candidate digit.

Therefore, this simplistic approach would suffer in terms of speed, especially on a low-end processor like the AVR. To mitigate these shortcomings, a series of optimizations are discussed on the algorithmic level.

Set-based constraint representation

Both the needless iterations and slow constraint violation check can be solved by transforming the problem into a constraint-based representation. Each row, column, and block will be represented by a set of legal digits, i.e. the digits that do not yet exist in the respective region. Of course, these sets are dynamic and will change during the solving process. Moreover, a binary 9x9 matrix is necessary to distinguish between the clue cells and the starting empty cells. This matrix is constant for a given puzzle. Altogether, this is the necessary and sufficient information to describe any puzzle. The representation is formally described below.

Let G be a 9x9 Sudoku puzzle, where $G_{ij} = \begin{cases} u \in [1,9], & \text{if cell } (i,j) \text{ is a clue} \\ 0, & \text{otherwise} \end{cases}$

We define the row, column, and block legal digit sets as

$$R_i = \{d \mid d \in [1,9] \ \nexists j \in [1,9] \ G_{ij} = d\} \qquad \forall i \in [1,9]$$

$$C_j = \{d \mid d \in [1,9] \ \nexists i \in [1,9] \ G_{ij} = d\} \qquad \forall j \in [1,9]$$

$$B_{kh} = \{d \mid d \in [1,9] \ \nexists i \in [3k-2,3k] \ \nexists j \in [3h-2,3h] \ G_{ij} = d\} \qquad \forall k,h \in [1,3]$$

respectively, and the binary matrix denoting distinguishing the non-clue (empty) cells as

$$E \in [0,1]^{9x9} \text{ where } E_{ij} = \begin{cases} 0, & \text{if cell } (i,j) \text{ is a clue} \\ 1, & \text{otherwise} \end{cases}$$

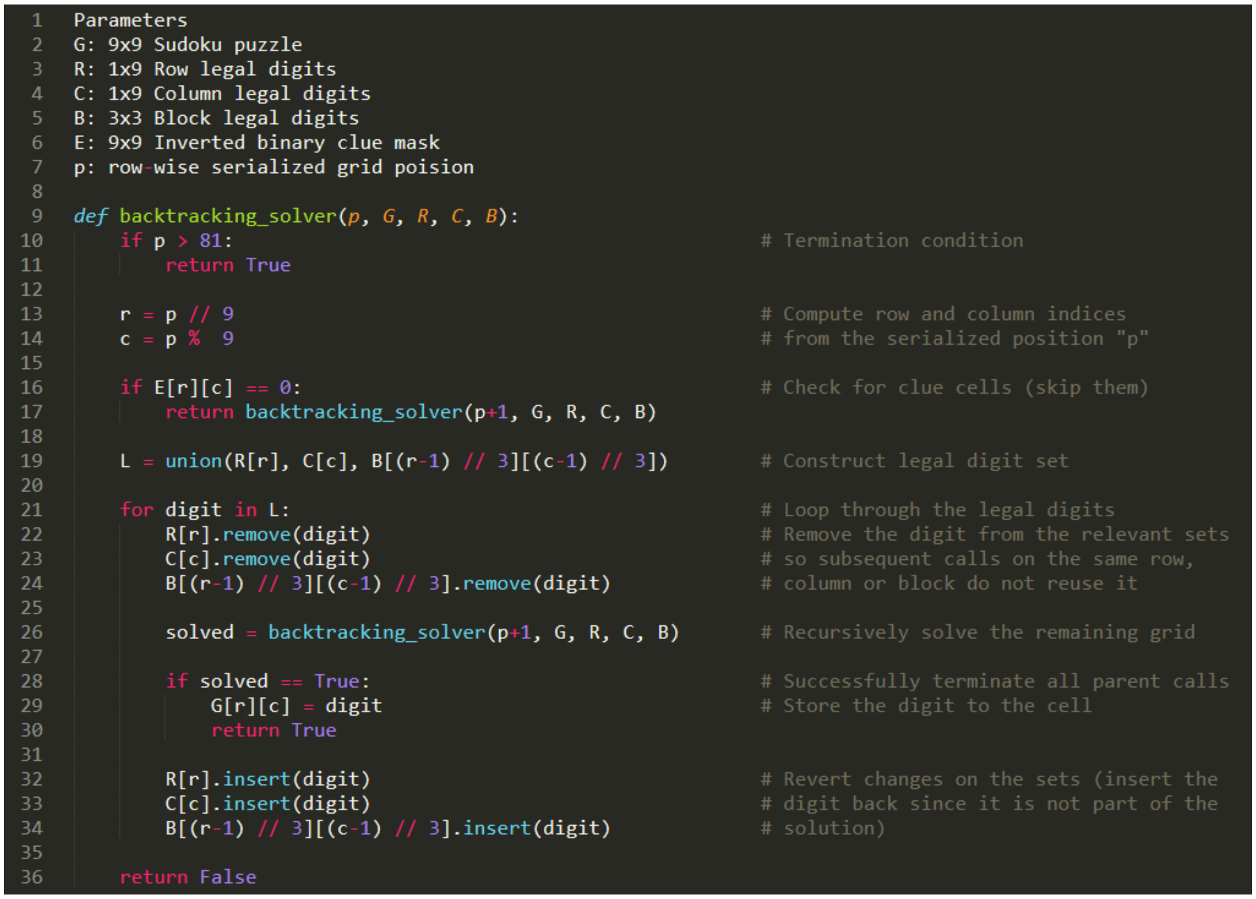
With this representation, the set of legal digits L for any non-clue cell at any time is given by the intersection of the aggregate sets of the structures, in which the cell belongs. In other words,

$$L_{ij} = R_i \cup C_j \cup B_{kh} \qquad \forall i,j \in [1,9]$$

where $k = (i-1) \text{ div } 3 + 1$ and $h = (j-1) \text{ div } 3 + 1$.

Rewriting the above pseudocode under this transformation, yields the code below.

Interestingly, this transformation removes the necessity to check for constraint violations altogether, for the simple reason that the candidate digits we iterate through emerged from the constraints. Of course, for this approach to be faster, we need an inexpensive way to perform operations on sets, namely insertion, deletion, and union.



```
Parameters
G: 9x9 Sudoku puzzle
R: 1x9 Row legal digits
C: 1x9 Column legal digits
B: 3x3 Block legal digits
E: 9x9 Inverted binary clue mask
p: row-wise serialized grid poision

def backtracking_solver(p, G, R, C, B):
    if p > 81:                                              # Termination condition
        return True

    r = p // 9                                              # Compute row and column indices
    c = p %  9                                              # from the serialized position "p"

    if E[r][c] == 0:                                        # Check for clue cells (skip them)
        return backtracking_solver(p+1, G, R, C, B)

    L = union(R[r], C[c], B[(r-1) // 3][(c-1) // 3])        # Construct legal digit set

    for digit in L:                                         # Loop through the legal digits
        R[r].remove(digit)                                  # Remove the digit from the relevant sets
        C[c].remove(digit)                                  # so subsequent calls on the same row,
        B[(r-1) // 3][(c-1) // 3].remove(digit)             # column or block do not reuse it

        solved = backtracking_solver(p+1, G, R, C, B)       # Recursively solve the remaining grid

        if solved == True:                                  # Successfully terminate all parent calls
            G[r][c] = digit                                 # Store the digit to the cell
            return True

        R[r].insert(digit)                                  # Revert changes on the sets (insert the
        C[c].insert(digit)                                  # digit back since it is not part of the
        B[(r-1) // 3][(c-1) // 3].insert(digit)             # solution)

    return False
```

Figure 2: Set-based recursive backtracking algorithm

Set and digit encoding

Again, we will consider the representation of the problem, which in this case is the sets of legal digits. If we consider these sets as bitmasks, with 9 bits, each of which represents a single digit, then the desired set operations are trivial to perform. Union is given by the bitwise conjunction of the individual masks, and insertion/deletion is achieved through bit toggling. In the case of insertions and deletions, any number of them within the set boundaries can be performed in parallel, in constant time. So far, we have eliminated the need for expensive constraint violation checks, as well as the certainly illegal iterations. But in the process, we increased the complexity of transitioning from one iteration to the next, since incrementing a counter is definitely simpler than extracting the set bits of a mask, one by one.

While the naïve backtracking method indeed benefits from its exceedingly simple transition step, we will prove that extracting the individual digits from the bitmask is not too demanding, computationally, if done properly.

> Suppose m is a bitmask, where the i-th bit is 1 if the decimal digit $i + 1$ exists in the set. If the digit itself is one-hot encoded as the value $1 \ll i$, then we can use the properties of two's complement to extract the rightmost set bit, inexpensively.

Since the two's complement is obtained by flipping all bits up to and excluding the rightmost 1, we observe that the bitwise conjunction of a non-zero m and its two's complement, $\widetilde{m}$ will always yield the encoded digit itself, $1 \ll i$. XOR-ing the mask m with the extracted digit, $1 \ll i$, removes (toggles) the digit from the mask and the process can continue for the next digit in m. The termination condition is m becoming 0 as a result of XOR. Note that the process is stable, since once m reaches 0, both m and d will always evaluate to 0.

Altogether, the following sequence of steps extracts and removes an encoded digit from the mask m:

$$
\begin{aligned}
&d = m \,\&\, \widetilde{m} \\
&m = m \oplus d
\end{aligned}
$$

Depending on the target platform, the above sequence is translated into a few assembly instructions.

The bit extraction process is presented below for the general case.

$$
\begin{array}{rcccccccc}
m = & b_{n-1} & b_{n-2} & \dots & b_{i+1} & \mathbf{1} & 0 & \dots & 0 \\
\widetilde{m} = & \overline{b_{n-1}} & \overline{b_{n-2}} & \dots & \overline{b_{i+1}} & \mathbf{1} & 0 & \dots & 0 \\
d = m \,\&\, \widetilde{m} = & 0 & 0 & \dots & 0 & \mathbf{1} & 0 & \dots & 0 \\
m = m \oplus d = & b_{n-1} & b_{n-2} & \dots & b_{i+1} & \mathbf{0} & 0 & \dots & 0
\end{array}
$$

Figure 3: One-hot encoded digit extraction process from legal digit bitmask

With this representation and encoding, we can leverage the benefits of the constraint-based approach, while maintaining a low-overhead loop without costly memory accesses to obtain the next digit in the set. Also, because the element-set operations can be parallelized for any number of digits, executing once for all digits simultaneously, the body of the loop can be further sped up by considering its temporal implications. The sequence of operations per iteration and across time for a single function call is shown in Figure 4.

Since XOR-ing a digit with the set mask toggles the digit's "presence" in that set, we can simultaneously remove one digit and insert another, if we know that the former exists in the set and the latter does not. Thus, in our case, the set operations are halved. The only added cost is the creation of a temporary set, holding both digits, which is obtainable by OR-ing the two digits, assuming the one-hot digit encoding discussed previously. Putting everything together yields the pseudocode in Figure 5.

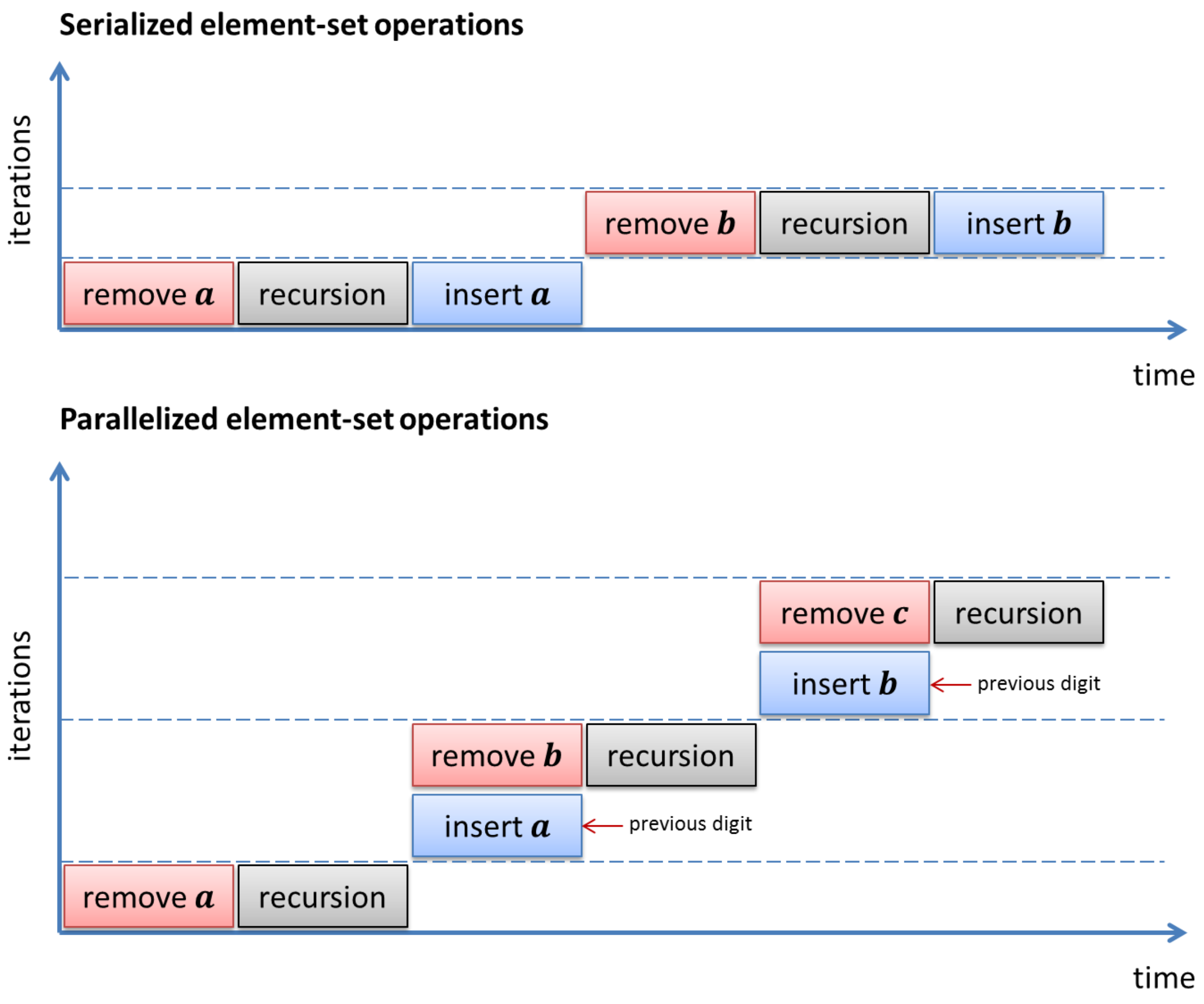
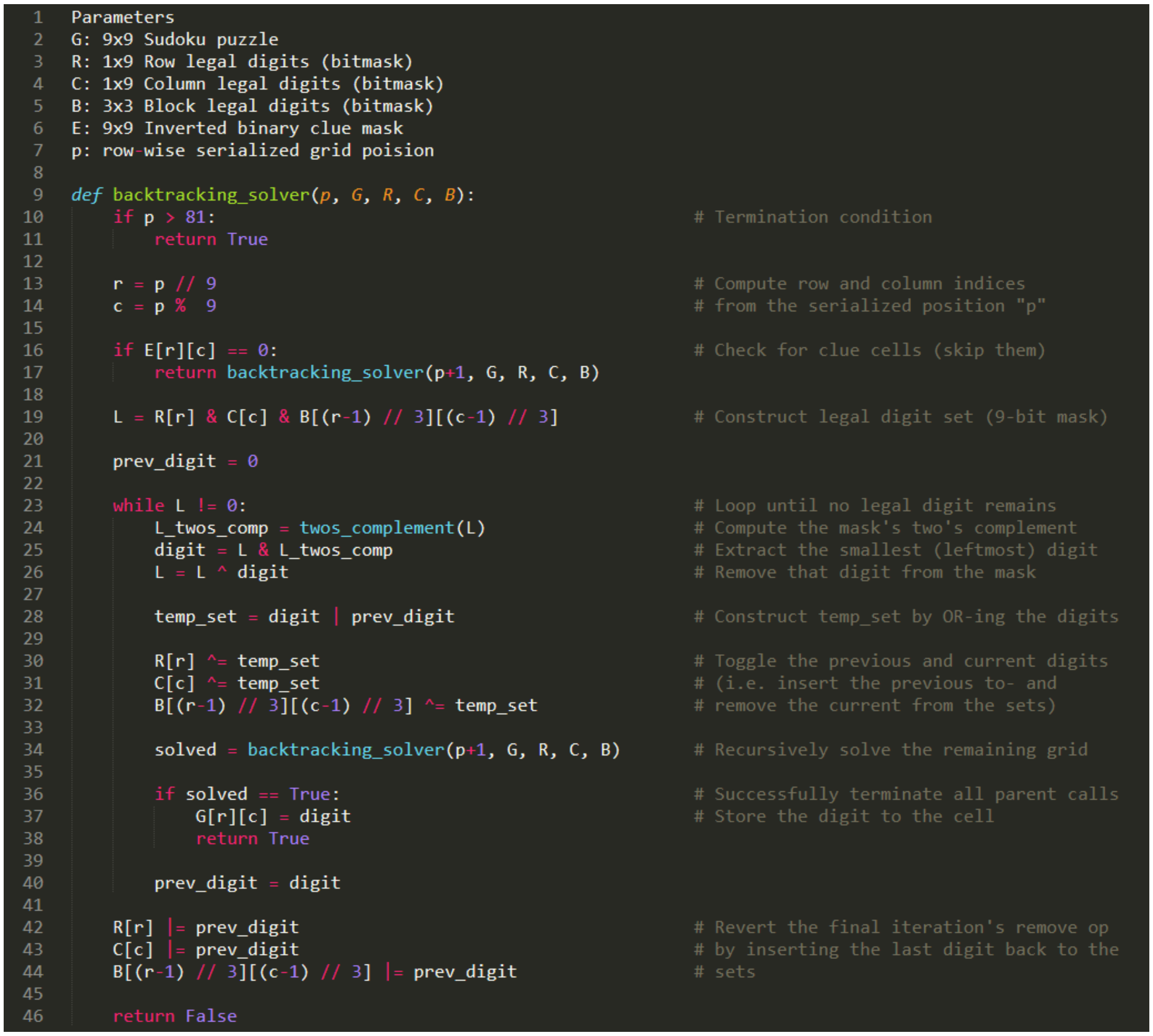


Figure 4: Serialized vs parallelized element-set operations



```
Parameters
G: 9x9 Sudoku puzzle
R: 1x9 Row legal digits (bitmask)
C: 1x9 Column legal digits (bitmask)
B: 3x3 Block legal digits (bitmask)
E: 9x9 Inverted binary clue mask
p: row-wise serialized grid poision

def backtracking_solver(p, G, R, C, B):
    if p > 81:                                      # Termination condition
        return True

    r = p // 9                                      # Compute row and column indices
    c = p %  9                                      # from the serialized position "p"

    if E[r][c] == 0:                                # Check for clue cells (skip them)
        return backtracking_solver(p+1, G, R, C, B)

    L = R[r] & C[c] & B[(r-1) // 3][(c-1) // 3]     # Construct legal digit set (9-bit mask)

    prev_digit = 0

    while L != 0:                                   # Loop until no legal digit remains
        L_twos_comp = twos_complement(L)            # Compute the mask's two's complement
        digit = L & L_twos_comp                     # Extract the smallest (leftmost) digit
        L = L ^ digit                               # Remove that digit from the mask

        temp_set = digit | prev_digit               # Construct temp_set by OR-ing the digits

        R[r] ^= temp_set                            # Toggle the previous and current digits
        C[c] ^= temp_set                            # (i.e. insert the previous to- and
        B[(r-1) // 3][(c-1) // 3] ^= temp_set       # remove the current from the sets)

        solved = backtracking_solver(p+1, G, R, C, B)   # Recursively solve the remaining grid

        if solved == True:                          # Successfully terminate all parent calls
            G[r][c] = digit                         # Store the digit to the cell
            return True

        prev_digit = digit

    R[r] |= prev_digit                              # Revert the final iteration's remove op
    C[c] |= prev_digit                              # by inserting the last digit back to the
    B[(r-1) // 3][(c-1) // 3] |= prev_digit         # sets

    return False
```

Figure 5: Bitmask-based recursive backtracking algorithm with one-hot digit encoding

Spatiotemporal locality

Another notable observation about the backtracking algorithm is the joint spatial and temporal locality of its steps. Let's assume the algorithm processes cells one-by-one from the top left Sudoku corner and in a row-wise fashion. Then, for any visited cell, it is highly likely that both it and cells in its serialized neighborhood will be visited again in the "near" future. This fact stems from the inherent consecutive order in which the cells are visited and the exponentially diminishing probability that rules will not be violated by some assignment in the next k cells the algorithm visits.

Therefore, regardless of the total number of iterations until the puzzle is solved, the algorithm will perform the majority of the backtracking steps within "small" neighborhoods of cells at "small" time intervals. There will of course be cases where long chains of consecutive unidirectional steps are formed, but in the majority of cases, these chains will be kept short. This simple fact will be heavily exploited via caching and lazy execution, to diminish the cost of memory accesses.

Recursion elimination

Lastly, the backtracking algorithm can be transformed from a recursive function to its equivalent iterative version. The transformation is trivial, by wrapping the entire function body with an infinite loop, replacing recursive function calls with continuation statements. Also, the return statements must be removed and the termination condition must break out of the outer loop. Far less trivial, however, is the appropriate management of the information that would reside in the function frames, as we transition from the automatic stack-based approach to a manual, centralized one. This nuance will not be further analyzed though, since the Assembly implementation will use a hybrid approach, where recursive calls are eliminated, but their stack is maintained.

The benefit of eliminating recursions lies in the reduction of function calls. Each function call, aside from its body, incurs a non-negligible extra cost for its frame. The frame contains the function’s arguments and the return address, all of which need to be pushed onto the stack prior to the call itself, thus taking up more CPU cycles. Aside from that, however, even in the case of functions without arguments, the instructions used to call- and return from- a function have a higher cycle cost, compared to jump instructions.

Implementation outline

Our approach is to implement the computationally demanding parts in Assembly, while the non-performance-critical functionality will be implemented in C, due to its significantly lower development cost. The entire communication protocol, the associated command actions, as well as the setup and initialization methods and the main loop are all written in C. Assembly is reserved for the Sudoku solver engine and a dedicated software interrupt which will act like a hook or a degenerate kernel and will provide seamless preemption and context switching capabilities between the engine and the main loop, in order to support the debugging commands, as well as the progress bar update, with minimal computational overhead.

A block diagram of the most important components is shown below. Both the USART and Timer1 ISRs trigger the software interrupt, which is in turn responsible for safely returning control over to the main loop, if the Sudoku engine was running and got interrupted by the hook. This mechanism is a tradeoff between implementing a full-fledged kernel or significantly slowing down the engine. Communication between the ISRs is achieved via shared variables.

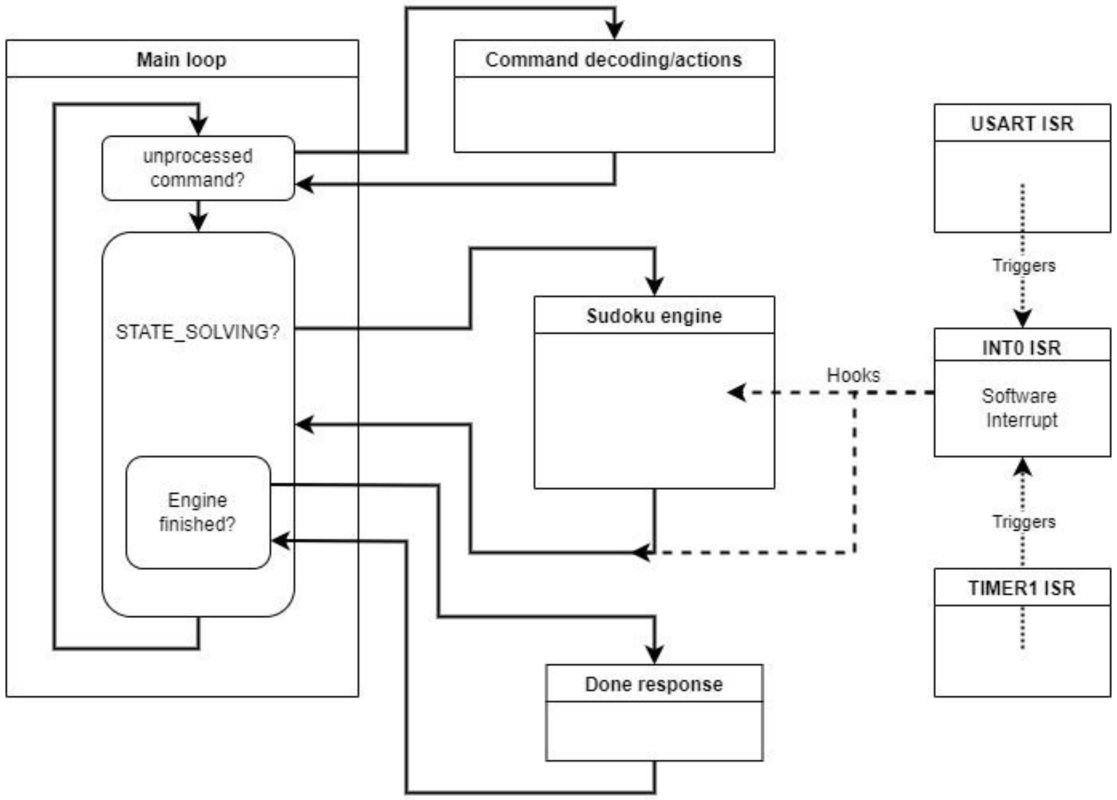


Figure 6: Implementation's top-level block diagram

Memory organization

Memory is allocated globally and statically for each of the major processes, namely main, USART, command decoding, INT0, and the Sudoku engine. Main reserves just 1 byte, for keeping track of the program's state. USART reserves 1 byte for the encoded received command and 3 bytes for the command's digits. Three bytes are reserved for command decoding, denoting whether the Sudoku data have been cleared and the row and column index of the last sent Sudoku digit of the solved grid. External interrupt INT0 reserves 35 bytes to save an "image" of the entire Register File, the status register and the program counter, so that the engine can resume its execution at a later time. INT0 also reserves 1 byte, denoting which actions the hook has been triggered to perform, namely *break* the engine's execution and *update* the progress bar.

The Sudoku engine reserves by far the most memory, holding a 9x9 grid to be filled when the computation is finished or preempted, and a 137 byte array holding its input data for a given puzzle. The input data array is sparsely used and with repeating elements, but it is through its structure and said sparsity that we can speed up the engine. Lastly, another byte is reserved for a guard variable, which contains two flags: one for denoting whether the currently executed code is the engine's and another for denoting that a saved, unused CPU "image" exists and therefore the engine requires restoration to its former state and not reinitialization.

The layout and semantics of the solver's image and the input memory locations are illustrated below.

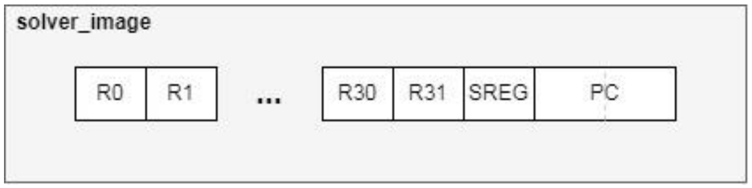


Figure 7: Solver image's memory layout

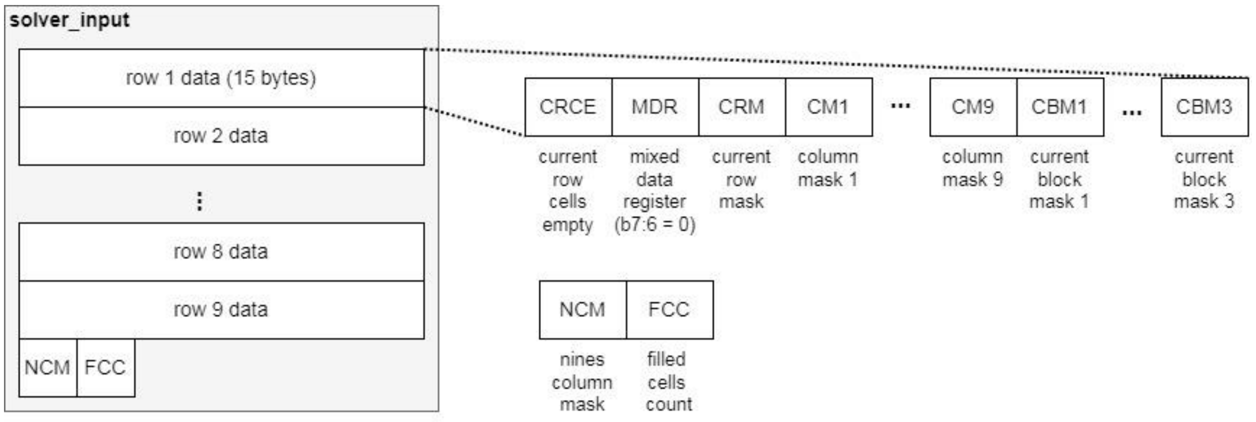


Figure 8: Solver input's memory layout and semantics

Resource utilization

The communication bus utilizes pins PD0 and PD1 for Rx and Tx respectively, as well as the USART_RXC_vect interrupt vector.

Timer/Counter1's channel A is used to update the progress bar at a 40Hz rate.

The software interrupt uses external interrupt 0 and therefore pin PD2.

The progress bar uses all 8 pins of Port C for its LEDs.

The entire code statically reserves 263 bytes of internal SRAM.

Register TCNT2 for simulating the UDR register during simulation.

USART Implementation

For USART to function according to the desired settings, multiple bits must be set to appropriate values across a number of control registers. The asynchronous mode is selected by setting the UMSEL flag of the UCSRC register to 0. The frame size is controlled by the UCSZ2 bit of the UCSRB register and the UCSZ1:0 bits of the UCSRC register and as per the datasheet, these bits must be set to 011 for 8-bit frames. For the parity bit mode to be disabled, UPM1:0 bits of the UCSRC register must be set to 00. The USBS bit of the UCSRC register must be set to 0, in order to have 1 stop bit. The UCPOL bit of the same register, which sets the relationship between data output change and data input sample, and the synchronous clock (XCK), must be set to 0, because we are using the USART in asynchronous mode. We set UBRRL to the value 0x64 based on the previous baud rate calculations. Finally, the receiver, the transmitter and the RXC flag interrupt must be enabled, by setting the RXEN, TXEN and RXCIE bits of the UCSRB register to 1.

The USART protocol's implementation is split into two separate processes: command reception and command execution. Command reception is handled in the USART's Interrupt Service Routine (ISR), where the received ASCII characters are processed and gradually form completed commands, or are rejected along with the current command-character sequence. This implementation allows the ISR to be short, as good practice dictates. When a command is finished, the program enters an action selection method in the main loop, where the appropriate actions are performed based on the received command. This way, communication works in full-duplex mode, since the transmission of a response from the chip does not block it from receiving new commands.

Command creation and recognition is based on a unified 8-bit encoding, with which all commands and all their intermediate states are encoded. The encoding is described below:

Bit	7	6	5	4	3	2	1	0
	LF flag	CR flag	CC3	CC2	CC1	CC0	BL1	BL0

- **Bit 7 – LF flag**
  Indicates whether LF has been received after CR.
- **Bit 6 – CR flag**
  Indicates whether CR has been received after a valid command code (CC).
- **Bit 5:2 – Command Code**
  Indicates whether a valid, an invalid or no command has been received from the serial port.
- **Bit 1:0 – Base length**
  Indicates the quantity $n - 3$, where n is the number of characters in a command. Base length indicates how many characters a command is made of, apart from the minimum of 3 characters.

All valid commands have the <CR><LF> ASCII character sequence as a common suffix. The two most significant bits (MSB) of the 8-bit command variable (LF flag and CR flag) will be set to 1 if the ending sequence <CR><LF> has been successfully received in the correct order. At that point, the current command terminates and is processed by the decoding process, while the ISR is free to receive new data. Technically, a buffer is required between the ISR and the command decoding process, to queue completed commands awaiting processing, but we consider this beyond the scope of this project.

The protocol defines 9 commands which the AVR can receive and since the commands are prefix free, there is no need for extra command codes to encode intermediate states. Thus, 4 bits for the command code are sufficient and provide some flexibility for future protocol extensions. However, we will reserve two more command codes, named NONE and INVALID, for the purposes of controlling the ISR's receiver FSM. The command code NONE indicates the absence of user input to the AVR microcontroller and of any received command awaiting processing. The command code INVALID indicates an erroneous sequence of characters. The assigned command codes are presented below:

Table 3: Command code allocation

Command	Command Code (CC)			
	Bit 3	Bit 2	Bit 1	Bit 0
NONE	0	0	0	0
INVALID	1	1	1	1
AT	0	0	0	1
C	0	0	1	0
N	0	0	1	1
P	0	1	0	0
S	0	1	0	1
T	0	1	1	0
OK	0	1	1	1
B	1	0	0	0
D	1	0	0	1

The two least significant bits of our command encoding format, correspond to the *base-length* of each received command. For example, the length of the AT<CR><LF> command is 4, so by definition, it's base-length will be 1. On the other hand, commands with only three characters (e.g. C, P, etc.), have 0 base-length. Since the shortest command has 3 characters and the longest has 6, just 2 bits are sufficient for the base-length.

The command code is assigned according to the first received character after an <LF> and in case the base-length exceeds 3, the received sequence is discarded and replaced by the INVALID byte. The combination of these mechanisms, together with the characteristics of the given protocol guarantee the detection of any error which results in an erroneous command character sequence.

Overall, the unified 8-bit command encoding is the following:

Table 4: Command encoding

Defined commands	Full byte	LF flag	CR flag	Command Code (CC)				Command base-length	
	hex	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NONE	0x00	0	0	0	0	0	0	0	0
INVALID	0x3C	0	0	1	1	1	1	0	0
AT<CR><LF>	0xC5	1	1	0	0	0	1	0	1
C<CR><LF>	0xC8	1	1	0	0	1	0	0	0
N<X><Y><VAL><CR><LF>	0xCF	1	1	0	0	1	1	1	1
P<CR><LF>	0xD0	1	1	0	1	0	0	0	0
S<CR><LF>	0xD4	1	1	0	1	0	1	0	0
T<CR><LF>	0xD8	1	1	0	1	1	0	0	0
OK<CR><LF>	0xDD	1	1	0	1	1	1	0	1
B<CR><LF>	0xE0	1	1	1	0	0	0	0	0
D<X><Y><CR><LF>	0xE7	1	1	1	0	0	1	1	0

When the serial port receives a character in the UDR register, the RXC flag is enabled, an interrupt occurs and the USART ISR code is executed. The character processing logic is implemented using an *if statement*, which is basically an FSM, in the ISR code. The character processing logic is separated into three parts. First, a CR or LF character check is performed. Depending on the value of the LF flag and the CR flag bits, the unified encoding format variable is updated with the appropriate bit shift. In the case of an erroneous LF character reception, the command will be reset into a NONE command. In any other erroneous case of an LF or CR reception, the command will be reset into an INVALID command. When receiving the first character of a command, the command variable will be initialized accordingly. In the case of an unexpected ASCII character, the command will be reset into an INVALID command. Finally, in the case of a command with more than one base characters, the appropriate actions will be executed as part of the USART protocol.

The action execution part of the USART protocol, as previously mentioned, is implemented separately from the ISR code. To establish a checking mechanism to distinguish the valid from the invalid commands based on the current state, we use an FSM. The only possible state transitions are when a *play* command is received or when a *DONE* response has been transmitted from the AVR.

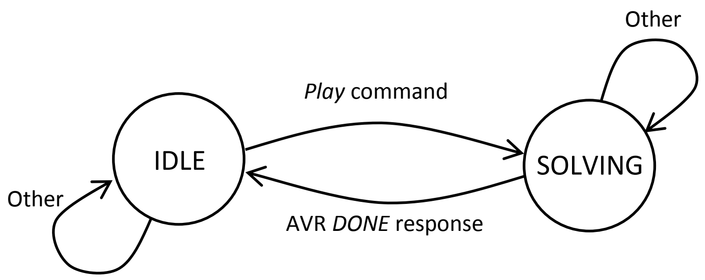


Figure 9: Program FSM

The moment a command (valid ASCII character sequence) has been received, the appropriate actions are executed. These actions are presented in the following table:

Table 5: Command actions

Command (from PC)	Action	Function call
AT<CR><LF>	_	transmit_ok_response()

C<CR><LF>	▪ Break if the FSM is in SOLVING STATE
	▪ Update clear flag to 1
		sudoku_clear()
		transmit_ok_response()

N<X><Y><VAL><CR><LF>	▪ Break if the FSM is in SOLVING STATE
	▪ If Sudoku data have not been cleared, set clear flag to 1
		(if sudoku_clear_flag == 0) sudoku_clear()
		sudoku_insert_value()
		transmit_ok_response()

P<CR><LF>	▪ Break if the FSM is in SOLVING STATE ▪ Update clear flag to 0 ▪ Change FSM state to SOLVING STATE	transmit_ok_response()
S<CR><LF>	▪ Break if the FSM is in SOLVING STATE ▪ Initialize last_sent_x/y variables	transmit_cell_digit()
T<CR><LF>	▪ Break if the FSM is in SOLVING STATE ▪ Update last_sent_x/y variables	(if last_sent_x/y >= 9) transmit_done_response() transmit_cell_digit()
OK<CR><LF>	–	–
B<CR><LF>	▪ Break if the FSM is in IDLE STATE ▪ Change FSM state to IDLE STATE	transmit_ok_response()
D<X><Y><CR><LF>	▪ Break if the FSM is in IDLE STATE	transmit_cell_digit()

Sudoku data representation

In any Sudoku memory location, except those for a USART commands ASCII parameters, digits are stored in a one-hot encoding, where the decimal digit dd is represented by $1 \ll (dd - 1)$. In the case of decimal 9, since the AVR is an 8-bit architecture, its representation is concatenated to the lower 8 bits and is therefore represented by 0x00 in the finished Sudoku cells.

The input data to the solver are the legal digit masks of the rows, the columns and the blocks, the cell-empty masks for each cell in a row and the starting filled cell count, i.e. the number of clues. Essentially, the input is the legal digit sets and the binary matrix denoting the non-clue cells described in the *Backtracking algorithm analysis* section. Naturally, since AVR is an 8-bit architecture and each mask is 9 bits, the data is organized in such a way that all the masks fit in memory and in registers, but the mask bits concerning the digit 9 and the column 9 are specially handled to fit in separate locations. The memory layout semantics are identical as those used in the Register File, and are therefore presented in the *Computational engine* section.

We should note, however, the specific locations where data is written in the solver_input array, before it is passed to the solver. The locations pertaining to row 1 data and the NCM and FCC bytes are all written. Apart from that, for each remaining row in solver_input, only the locations pertaining to CRCE, MDR and CRM are written, while the column masks are left unmodified and thus contain garbage data. The block masks are written at every 3rd row. The remaining locations pertaining to block masks are unused, yet allocated to achieve regularity between rows. As for the unwritten column masks for the remaining rows, these locations will be written by the engine during puzzle solving. The idea is that instead of having to undo the previous changes in the column masks, we can simply get the column mask of the previous row and update it based on the digit assigned to the respective cell in the current row. We are trading off memory space to cut down on computations.

Computational engine

The Sudoku engine consists of three parts: the initialization or image restoration, the backtracking algorithm, and the result extraction and stack reset. Of those parts, the implementation of the backtracking algorithm is the most interesting and challenging. This is also the only segment of the engine where preemption is allowed. The reasoning is that the other two phases are not too time-consuming, so we reserve the preemption ability for the main algorithm.

Initialization / restoration

The first phase prepares the Register File to either start solving a new puzzle or continue where the previous call left off. In the initialization case, all registers are cleared, except for NCM and FCC which are fetched from the solver_input array, and the Z register, which is initialized to point to the solver_input data. Traditionally, when an address register is meant to address an array, its value is initialized to the location of the first element of the array. In this implementation, however, the register is initialized to an address before the start of the array (with an offset of 30 bytes, or 2 rows worth on solver input data). The reason for this offset is twofold: the algorithm will increment Z by 15 bytes immediately when it starts, and we want to have a specific window of the data “in view”. The latter point will become clear in the *Lazy & static memory access* section.

In the image restoration case, the old PC address is read from the image array and pushed to the stack, low-byte-first (i.e. the high-byte is on the top of the stack). Then, the status register and the entire Register File is restored to its old values, held in the image array. Finally, a *ret* instruction is executed to load the PC with the pushed address. This is the only way in the AVR ISA to jump to a location in code, without using any registers for that jump. If the *ijmp* instruction was used, the Z register would need to contain the jump location and would thus not be possible to restore it to any other value.

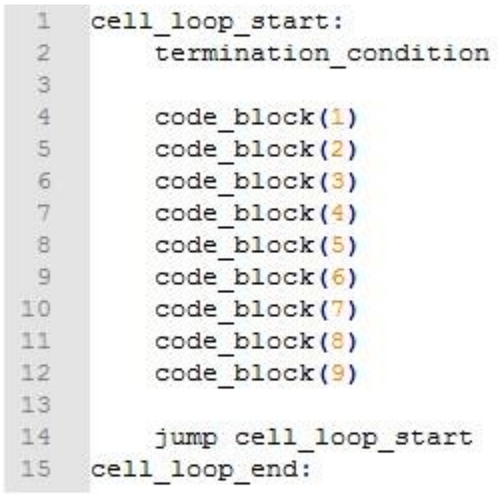
Loop unrolling

As discussed in the *Recursion elimination* section, it is beneficial to implement the backtracking algorithm iteratively, rather than recursively. This fact becomes apparent merely by examining the AVR ISA. The *rcall* instruction takes 3 cycles to complete and the *ret* takes 4. Comparing that against the 2 cycles for the *rjmp* instruction, we see that the iterative implementation enjoys a slight performance boost over the recursive one, before we even take argument-passing into account. An important note here is our disregard for any branch instructions controlling the jump instruction's target. While the jump control overhead is non-negligible, we do not include it in the discussion for the sake of simplicity and brevity. It suffices to say that in this implementation the benefits of using jumps instead of function calls outweigh the added control overhead.

Furthermore, the iterative approach incurs a loop overhead on every iteration. We can amortize this overhead by unrolling part of the loop, thus increasing the amount of "effective" iterations performed for the same overhead cost. This is a well-known, universally-used compiler optimization technique that is indeed beneficial, albeit with diminishing returns as the iteration body increases, since the overhead to useful code ratio decreases. As it so happens, the body of our cell loop will be considerably larger than the loop overhead, therefore it seems this technique will not be notably fruitful.

Indeed the amortization of the loop overhead is not the driving force behind this optimization. The key benefit of unrolling the cell loop is the **distinct** code segments or code blocks used inside the loop. In a traditional loop, the same code gets executed in every iteration and therefore uses the same CPU resources in every iteration. But, functions – or, in this case, loop bodies – that do not require many registers are bound to utilize only a fraction of the hardware resources, regardless of the quality of the code. If the loop is unrolled, however, copies of the loop body must be created in the source code. These copies suffer no restriction whatsoever to be identical, as long as the functional requirements are met. Consequently, we can alter – through the syntactic sugar of a macro – the operands and/or instructions of each code block, such that in the scope of a single iteration of the partially unrolled loop, the full hardware resources of the CPU are utilized, either directly (as read or modified registers) or indirectly (as caches to be taken advantage of through spatiotemporal locality).

The exact number of unrolled iterations cannot be arbitrarily selected, since we need to allocate a finite number of registers into these unrolled iterations, excluding registers used "globally" by all iterations. After careful consideration, it turns out that unrolling 9 iterations of the cell loop achieves a near 100% register utilization and, with a careful design, this particular unrolling offers highly desirable properties, as discussed in the following sections.


```
cell_loop_start:
    termination_condition

    code_block(1)
    code_block(2)
    code_block(3)
    code_block(4)
    code_block(5)
    code_block(6)
    code_block(7)
    code_block(8)
    code_block(9)

    jump cell_loop_start
cell_loop_end:
```

Figure 10: Partially unrolled, well-structured cell loop

Cell loop elimination

Based on what has been discussed so far, the cell loop would look – very abstractly – like the snippet in Figure 10. In a well-structured loop, an apparent source of inefficiency arises when we consider what happens during a backtracking step at any code block i. The flow at that code block would have break out of the remainder of the loop's body, enter a new iteration and – at an additional cost – immediately jump to the $(i-1)$-th code block. Such a program flow is highly inefficient and impedes the general goal of optimizing for performance.

Instead, we can abandon the idea of a well-structured loop and approach this subject with a hardware-leaning mindset. The equivalent of the cell loop can be designed as a finite state machine with 9 "composite states", i.e. code blocks. With this FSM-inspired design of Fig., we can jump to neighboring code blocks, at their desired points, with no additional overhead. The downside is the increased design and debugging complexity, which in this case, is an acceptable tradeoff for performance. Forward steps are significantly simpler, since they always jump at the start of the code block.

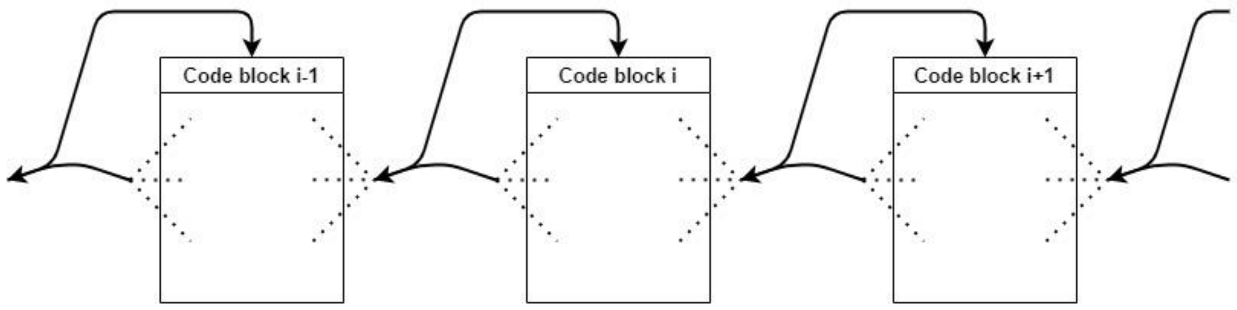


Figure 11: FSM-inspired flow between code blocks

Evidently, the forward steps can happen from many lines of code to a single location. For the backtracking steps however, a many-to-many relation will be needed, as we will see later on in the block diagram of the code block. Naturally, all other things being equal, the fewer the jump cases, the simpler and faster the code will be.

Cyclostationary register stack

We will exploit the structure of the distinct code blocks to take advantage of most of the Register File and use it as a cache, thus benefiting from the spatiotemporal locality of the algorithm, as explained in the homonymous section. Each code block will be assigned a column, such that all cells of that column in the Sudoku grid are processed exclusively by that code block. In other words, cells in column i are exclusively processed by code block i. Therefore, data is processed in a stationary manner, with respect to the relation between columns and code blocks. Furthermore, since the algorithm takes a single step at a time, in either direction, and there is wrap-around on the first and last code block, the process is cyclical.

We will consider each code block as an instance of the backtracking function. The absolute minimum information necessary to keep track of the progress in the digit loop is the encoded digit itself. Also, we want to keep track of the legal digit bitmask for each digit loop (i.e. for each code block), to avoid recomputing it in each digit iteration. Thus, 2 bytes per code block are sufficient as "local variables", to efficiently keep track of their progress. So, by allocating two registers per code block, we can retain the “local variables” of the code blocks in the Register File, thereby eliminating the need for constant memory accesses.

Essentially, we are creating an unconventional stack in the Register File which acts as a cache for the SRAM. When the register stack is full, the data on the “bottom” of the stack gets pushed to the SRAM stack and new data are written on the respective registers. The term *cyclostationary* is introduced to describe the properties of the register stack, as explained previously. An example of the register stack and its cooperation with the SRAM stack, in the context of code blocks, is illustrated in Figure 12.

There are several benefits of this approach. Firstly, cached cells (“function instances”) benefit from a 0-cycle access time of their data. Secondly, the algorithm uses the SRAM stack only when absolutely necessary. Thirdly, because of spatiotemporal locality, the data which is most likely to be reused in the future exists in the register stack. Lastly, this system eliminates the need for the candidate digit of each cell to be explicitly written to some location in SRAM, simply because that information exists – in some way – in the register and SRAM stacks.

This slew of advantages bears the overhead of controlling the register stack. Namely, the code must be able to decide when to push “frames”, i.e. the mask and digit, to the stack and when to pop them from it. If we examine the steps in Figure 12, we conclude that during forward steps the code either pushes “frames” onto the SRAM stack or it simply overwrites the respective code block registers, while during backtracking steps, the code either pops “frames” from the SRAM stack or it simply overwrites the respective registers. Therefore, it suffices to know, at any point, which code blocks must interact with the stack and which should not. The exact stack action, i.e. push or pop, will be unambiguously determined by the algorithm’s step, i.e. forward or backtracking, which is, in turn, specified by the specific code (branches) executed.

All the necessary information regarding the code block’s interaction with the stack can be represented by 9 bits, named *current-row-valid data*, or *CRVD*. When a CRVD bit is 1, the respective “frame” is row-valid, meaning that it pertains to the current row, even if the data have since been made obsolete (garbage). When a CRVD bit is 0, the code must interact with the SRAM stack upon transitioning to the respective code block. The CRVD bits are changed only immediately after an interaction with the SRAM stack, where the respective bit is set to 1, or at the transitions CB1 $\rightarrow$ CB9 and CB9 $\rightarrow$ CB1, where all bits are flipped. The CRVD bits are also shown in Figure 12, alongside the effects of the mechanism.

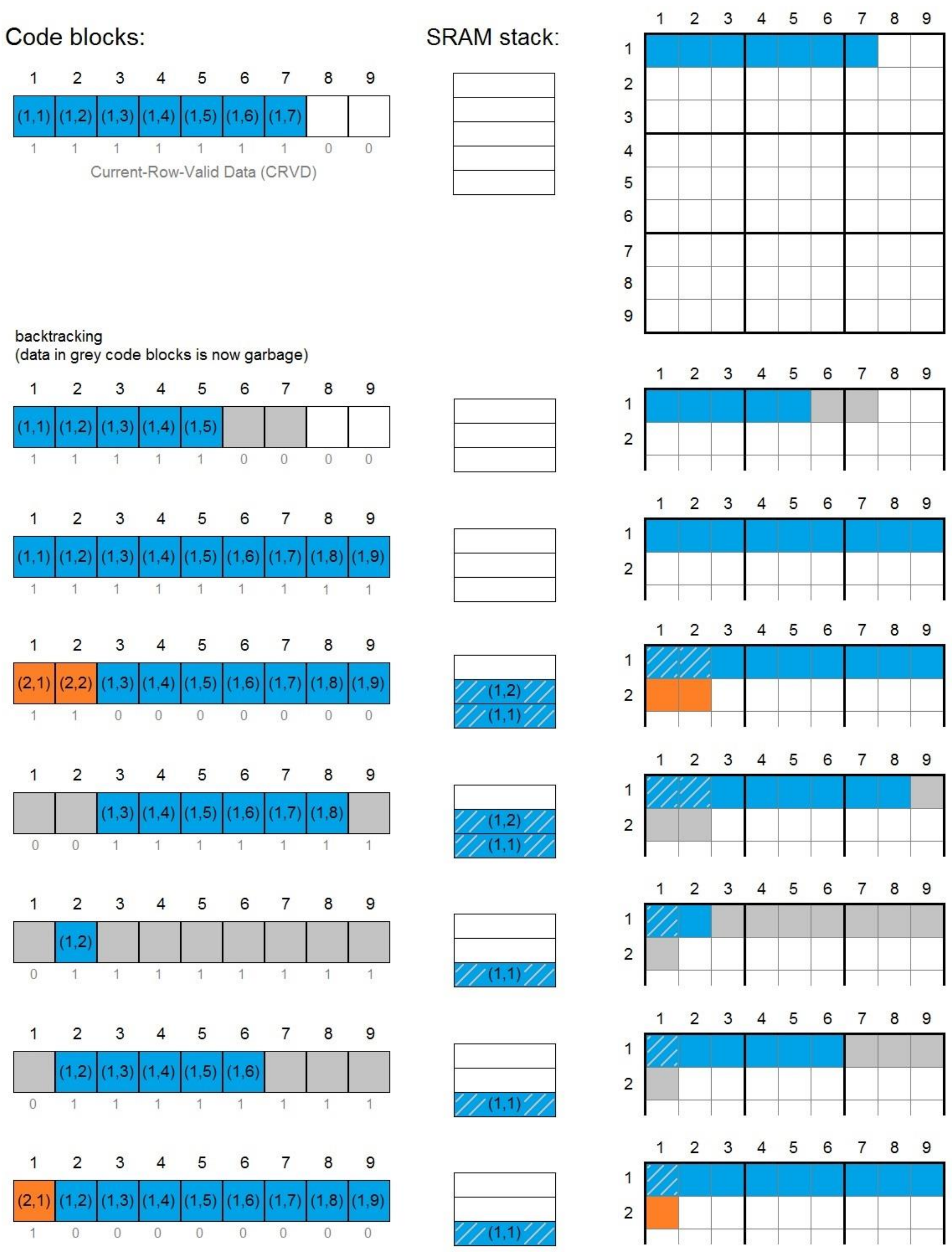


Figure 12: Abstract example of the register & SRAM stacks and their connection with the Sudoku grid, at various time points

Register allocation & mask caching

Already, 18 registers have been allocated to the code blocks for storing their “local variables” and 1 register is needed for the CRVD bits. Another 2 registers (the Z register) are necessary for accessing SRAM. We will reserve 1 more register for the filled cells counter, which due to its constant usage, we want to keep in the Register File. Lastly, a temporary/scratchpad register is unavoidable since we’re implementing the combined set operations described in the *Set and digit encoding* section. The remaining 9 registers must be allocated in a way that optimizes the performance of the algorithm, i.e. minimizes the execution cycles. Also, we need to keep in mind that, since the masks and the CRVD data are 9 bits wide, the 9th bits will have to be stored in an additional register.

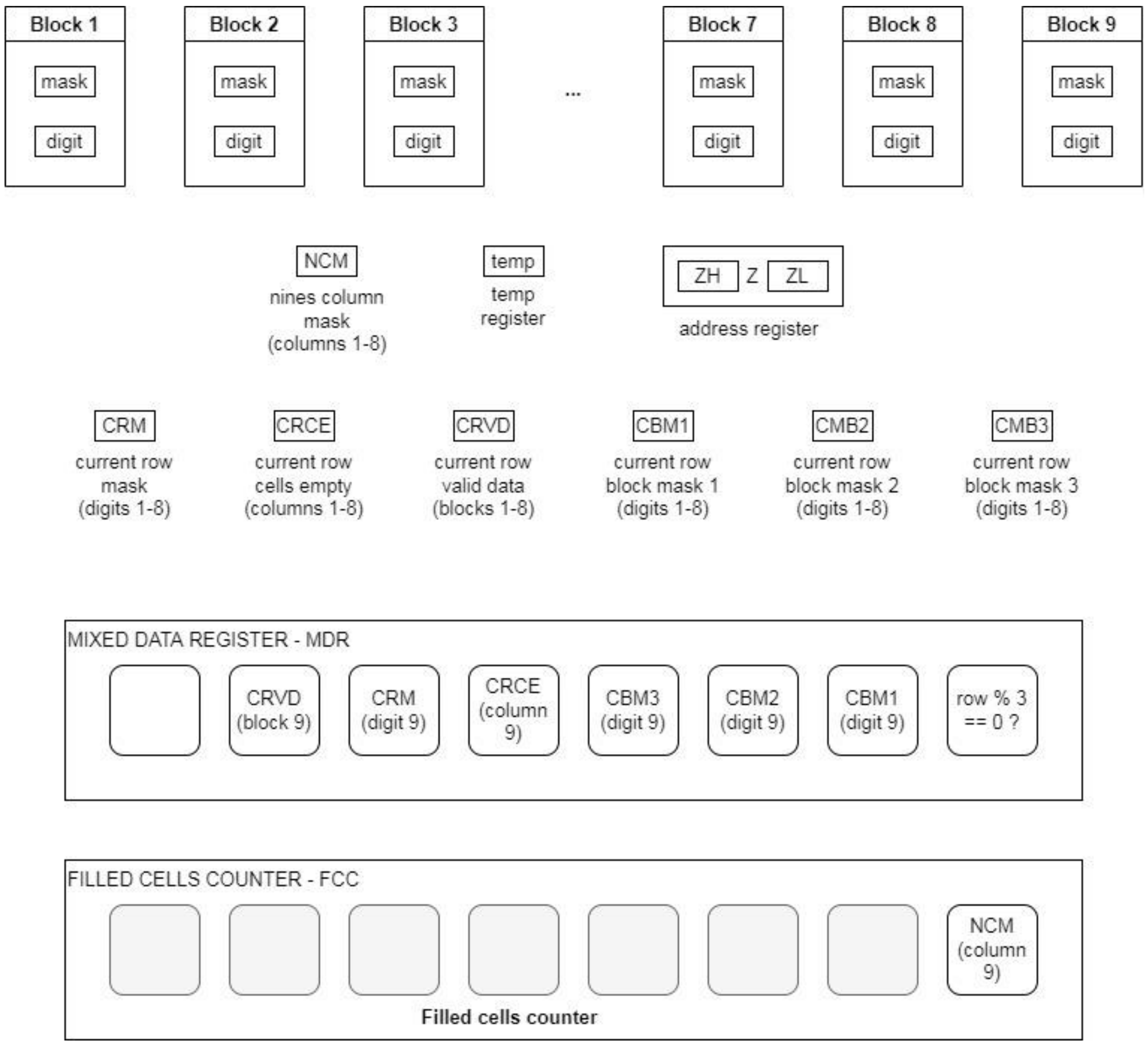

Block 1
mask
digit
Block 2
Block 3
...
Block 7
Block 8
Block 9
NCM
nines column mask (columns 1-8)
temp
temp register
ZH Z ZL
address register
CRM
current row mask (digits 1-8)
CRCE
current row cells empty (columns 1-8)
CRVD
current row valid data (blocks 1-8)
CBM1
current row block mask 1 (digits 1-8)
CMB2
current row block mask 2 (digits 1-8)
CMB3
current row block mask 3 (digits 1-8)
MIXED DATA REGISTER - MDR
CRVD (block 9)
CRM (digit 9)
CRCE (column 9)
CBM3 (digit 9)
CBM2 (digit 9)
CBM1 (digit 9)
row % 3 == 0 ?
FILLED CELLS COUNTER - FCC
NCM (column 9)
Filled cells counter

Figure 13: Register file allocation & semantics

The remaining registers can be used for caching the legal digit bitmasks, which will constantly be needed by the algorithm. Since the cells are traversed row-wise, the rational choice for which masks to cache are the row and block masks, for Sudoku band where the given row belongs. Indeed, the row mask will be the same for all cells of the row, thus it will be the most frequently used mask. The block masks will be the same for a third of the cells in the row and for three rows, thus will be the second most frequently used masks. The least frequently used masks, for some time interval, will be the column masks and therefore should not be cached by virtue of this cost-benefit argument.

So, 1 register is allocated for the current row mask and 3 registers are allocated for the current-row block masks. Additionally, the information whether a cell is a clue also takes the form of a bitmask, which is cached due to its high frequency usage in the CRCE register. Here, the word *empty* means *non-clue* and refers to the status of the cell in the given puzzle, it is not related to the runtime status of the cell.

Lastly, since the decimal digit 9 is not accommodated in the 8-bit legal digit masks, that information is separately stored in the NCM register. Note that in all other legal digit bitmasks, the position of the bit encodes the magnitude of the decimal digit, whereas in the NCM register, all bits refer to decimal 9 and the position of the bit describes the column to which it belongs.

The 9th bits of the various masks are grouped together in the – aptly named – MDR register, with one such bit being placed in the FCC register. This placement was originally decided with the intention of optimizing branch conditions, by copying the MDR data into SREG and then using conditional branches without further bit copies. For this purpose, the most significant bit of SREG would need to be unmodified (or set to 1), as it pertains to the global interrupt enable. This idea, however, did not work out and the peculiar bit placement remained due to legacy. The first bit of the MDR denotes whether the current Sudoku row is the first row of a band (i.e. row 1, 4, or 7 as depicted in Figure 12). In such case, code block 1 needs to load the CBMx data into the registers from memory.

Lazy & static memory access

Since the column masks are not cached, for each cell in each column, the mask must be read, modified and written back to memory for the rest of the cells to read. To reduce unnecessary mask modifications, this process happens only before a code block pushes a "frame" to the SRAM stack. The reason is that the modified column mask will definitely be needed immediately after the push (to create the mask of the new "frame") and is definitely **not** needed before that point. This optimization is a form of lazy execution and saves us from performing modifications which will be overwritten (due to backtracking) before being used even once.

Furthermore, the least costly, with respect to CPU cycles, way to perform the mask modification is to keep an entry of the column mask for each row. That way, when a cell changes its candidate digit due to backtracking, it can create the new column mask by reading the column mask of the previous row and removing the new digit from it, without having to place the old digit back. Again, we are trading off memory to save computations. This optimization is beneficial only if accessing the mask of the previous row takes the same time as accessing the column mask of the current row.

Any memory access incurs a minimum 2-cycle cost for reading or writing a byte, on top of the cost of setting the address in the Z register. The only way to access data from a previous row of the solver_input array without having to modify the Z register, is to set it to always point at the starting location of the previous row. That way, we can access any byte of the current or previous location via the displacement instructions, which access any location in the range $[Z, Z+63]$ without modifying Z and take 2 cycles to complete. Each row of the solver_input array contains 15 bytes, so we are well within the addressing range of the displacement instructions and can thus benefit from the column mask modification optimization.

Altogether, a high-level block diagram of the code block is presented in Figure 14. The many-to-many relation of the backtracking transitions mentioned in the *Cell loop elimination* section is clearly evidenced.

Solved board extraction

Finally, when the puzzle is solved, the cell digits must be written to the designated SRAM array, i.e. the sudoku_cells array. The digits of the last Sudoku row will exist in the Register File, while for all other cells the digits will reside in the stack. Thus, the “frames” are popped one-by-one from the stack and the digits of the non-clue cells are written in the sudoku_cells array.

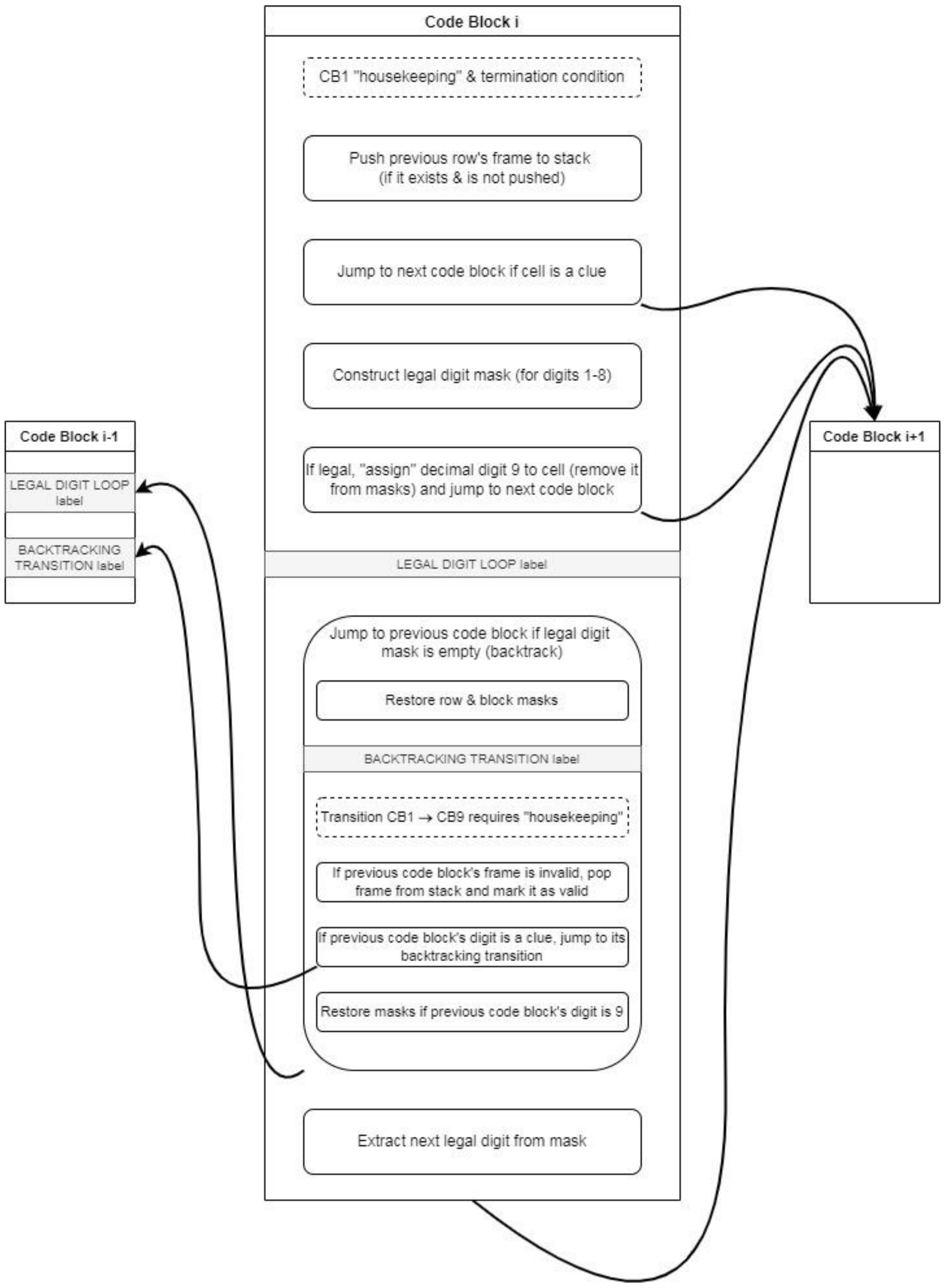

Code Block i
CB1 "housekeeping" & termination condition
Push previous row's frame to stack (if it exists & is not pushed)
Jump to next code block if cell is a clue
Construct legal digit mask (for digits 1-8)
If legal, "assign" decimal digit 9 to cell (remove it from masks) and jump to next code block
LEGAL DIGIT LOOP label
Jump to previous code block if legal digit mask is empty (backtrack)
Restore row & block masks
BACKTRACKING TRANSITION label
Transition CB1 → CB9 requires "housekeeping"
If previous code block's frame is invalid, pop frame from stack and mark it as valid
If previous code block's digit is a clue, jump to its backtracking transition
Restore masks if previous code block's digit is 9
Extract next legal digit from mask
Code Block i-1
LEGAL DIGIT LOOP label
BACKTRACKING TRANSITION label
Code Block i+1

Figure 14: Block diagram of Solver's code block

Software interrupt / hook

The AVR architecture offers external interrupts, which are interrupts triggered by signals passed to certain pins of the microcontroller. A, perhaps, lesser known property of these interrupts is that their function depends on the direction the pin is set to, i.e. input or output, via the DDR register. The pins can be configured as outputs, in which case the respective interrupts will fire based on the data written to the relevant port register. This mechanism provides a neat way of generating software interrupts, which are interrupts triggered by code, rather than external hardware signals.

The software interrupt interfaces with the USART and Timer1 ISRs via the hook_action shared variable. Bit 0 of this variable denotes the *break* action, where the hook stores the processor's "image", i.e. a copy of all general purpose registers, the status register and the PC in the solver_image array. Then, the hook pops the interrupt's return address from the stack and replaces it with the address of the end of the backtracking algorithm. With this mechanism, it is possible to preempt the execution of the Sudoku engine, go back to the main loop, act on received commands and send responses, while maintaining both a way for the solver to continue where it left off and a low computational overhead.

This technique of interfering with the interrupt's return address is potentially error-prone, if the conditions of its applicability are not clear or guaranteed. This technique only works if interrupt nesting, i.e. an interrupt interrupting another interrupt, is **not** allowed. This is the default case for AVR architectures, and unless an ISR enables interrupts inside its code, there is no potential for undefined or unexpected behavior. In any case, caution must be exercised to interfere with the return address in such a way that the desired behavior is generated.

Bit 1 of the hook_action controls whether the number of filled cells at the time the software interrupt fired will be read, processed and used to update the progress bar in port C.

The ISR performs the requested actions if and only if the solver's guard shared variable has its 0-th bit set, meaning that the code the ISR interrupted was the Sudoku engine's. If this condition is not met, the hook_actions variable is cleared and the ISR returns. The guard variable and this mechanism is put in place to ensure that the software interrupt hooks the Sudoku engine's code and not any other code.

Progress bar

The Timer/Counter1's ISR fires with a frequency of 40Hz, higher than the 30Hz requirement, and does not directly update the progress bar. Instead, it sets the *update* flag on the shared variable *hook_action*, and triggers the software interrupt by setting PD2 to low. Thus, when the Timer/Counter1 ISR finishes, the INT0 ISR will fire and update the progress bar.

Because the STK-500's LEDs are in common anode configuration, the data written to PORTC must be inverted, i.e. a logic 1 turns the LED off and a logic 0 turns it on.

USART Simulation

The USART simulation process involves the use of stim files and breakpoints for the ISR routine and the action execution method during code execution. When code execution is paused at a breakpoint, the appropriate register and memory values of the ATmega16 chip can be monitored and checked. The stim files execution, using the Microchip Studio IDE simulator, simulates the reception of different user commands from the PC to the AVR through the serial port of the chip. The proper functioning of the communication protocol is verified once the simulation is successful. That is when the AVR calls the necessary functions and sends the appropriate responses to the PC, based on the user input provided. These responses are recorded in a log file, which is created by the Microchip Studio IDE simulator. The simulation results include both the log files and the updated values of the AVR's registers and memory data space.