Machine Problem 1 Documentation

John Carlo A. Sumabat

November 29, 2020

Abstract

The following program is a simple checkers solver made in MIPS assembly language in partial fulfillment of requirements in CS 21: *Computer Organization and Architecture* under Jerome Beltran and Wilson Tan, 1st Semester S.Y. 2020–2021.

1 Introduction

The game of checkers involves an 8×8 board with two opposing players, and a mix of pieces that are either *kings* or *men*, whose rules will not be described here. The purpose of the program is to check whether the game can be won in one move, given a configuration of the board, where the term *move* denotes a series of one or more enemy captures.

The chosen implementation for this program is Implementation C: checking whether a solution exists in a board where men and kings can appear and printing the series of moves that leads to victory. A simple recursive backtracking algorithm was used for the relevant functions `jump` and `kjump`. Minor optimizations were employed to reduce stack usage and runtime.

2 Data Segment

Code Block 1 describes the contents of the data segment. All saved data except `yes` and `no` are initialized to zero.

Code Block 1 Data Segment

```
.data
  arr:     .byte 0:66          # array containing the board
  color:   .byte 0:1           # player color
  enemies: .byte 0:2           # array containing enemy pieces
  kingrow: .word 0:1           # row number of enemy's king row
  cur:     .word 0:1           # number of enemies captured
  moves:   .byte 0:385         # array containing winning moves
  yes:     .asciiz "\nYES\n"   # string to print when solution found
  no:      .asciiz "\nNO"      # string to print otherwise
```

- `arr` contains the 64 tiles of the input board in character bytes plus `\n\0` at the end. Hence, 66 bytes are allocated.
- `color` contains a character denoting the player to move, while `enemies` contains the enemy pieces. For example, if `color == ['W']`, then `enemies == ['b', 'B']`.
- `kingrow` and `cur` each hold a single integer.

- A move is described in six characters (e.g., `e3 g5\n`). Thus an estimate of $64 \times 6 = 384$ bytes plus one extra for a `\0` is more than enough allocated space to avoid writing to unallocated memory. This can be lowered to $30 + 1$ bytes since each test case has a maximum of 6 pieces and thus a maximum of 5 jumps, but there is no incentive to do so.

3 Initialization

The code responsible for preparing global variables are located under labels prefixed with `main`.

3.1 Accepting Input

Code Block 2 describes how the board is read and stored.

Code Block 2 Reading the board configuration

```
main:
  la      $a0, arr              # load arr address
  li      $a1, 10               # set character limit
  li      $v0, 8                # read string
input:
  syscall
  addi    $a0, $a0, 8           # get address of next 8th character
  addi    $t0, $t0, 1           # increment counter
  bne     $t0, 8, input         # read 8 rows
```

The loop variable is `$t0` which increments 8 times. Every loop, the address `$a0` of `arr` where the input is written is increased by eight. With this, the rows will be read as 8 input strings, each with 8 characters. Although each input string has `\n\0` at the end, it will be overwritten by the next string, unless it is the last iteration wherein it will be retained.

Next, the input indicating the player to move is read, as shown in Code Block 3.

Code Block 3 Reading the player character

```
  li      $v0, 12               # get character input
  syscall
  move    $s0, $v0
  sb      $s0, color            # color to move
```

After the procedure, the color of the player to move is stored in `$s0`. A pseudocode equivalent of the procedure is shown in Procedure 1.

Procedure 1

procedure $\textsc{Input1}(s_0 \ldots s_7, c)$
  for $i \leftarrow 0$ to 7 **do**
    $arr[i] \leftarrow s_i$
  $color \leftarrow c$

3.2 Saving Enemy Details

The `if` statement in Code Block 4 stores the enemy pieces and the location of the enemy's king row.

Code Block 4 Updating `enemies` and `kingrow`

```
main_if1:
  bne     $s0, 'W', main_else1  # if color is white:
  li      $t0, 'b'              # set 'b' and 'B' as enemies
  sb      $t0, enemies
  li      $t0, 'B'
  sb      $t0, enemies+1
  li      $t0, 0
  sb      $t0, kingrow          # set king row as 0
  j       main_endif1
main_else1:                     # else:
  li      $t0, 'w'              # set 'w' and 'W' as enemies
  sb      $t0, enemies
  li      $t0, 'W'
  sb      $t0, enemies+1
  li      $t0, 7
  sb      $t0, kingrow          # set king row as 7
main_endif1:
```

For example, if `$s0 == 'W'`, then the black pieces are stored in `enemies` and `kingrow == 0`. The opposite is true otherwise. Since the operation is fairly simple, only one register `$t0` is used for all instructions. A pseudocode equivalent can be seen in Procedure 2.

Procedure 2

procedure INPUT2(*color*, *enemies*)
  if *color* = 'W' **then**
    enemies[0] ← 'b'
    enemies[1] ← 'B'
  else
    enemies[0] ← 'w'
    enemies[1] ← 'W'

3.3 Counting Enemies

The program traverses `arr` to count the number of enemies, as shown in Code Block 5.

- Lines 1–9 and 25–30 represent a `for` loop. `$t0` holds the iterator variable `i` for the outer loop, while `$t1` holds the variable `j` for the inner loop. The `bge` instructions at lines 3 and 6 dictate that each loop will repeat 8 times. Using the formula `idx = col size * i + j`, the current index is obtained and stored in `$t2`. Then the element of `arr` at `idx` is stored in `$t3`.
- Lines 10–24 represent an `if` statement that adds 1 to `$s2` every time an enemy is encountered. This is done by checking which color to move (`$s0`) to branch to the correct label, then checking whether the current tile (`$t3`) is an enemy.
- Line 31 resets `$v0` to zero.

Procedure 3 provides pseudocode to aid understanding.

Code Block 5 Updating enemy count

```
  li      $t0, 0                    # int i = 0
main_forloop1:                      # for (i = 0; i < row size; i++)
  bge     $t0, 8, main_endforloop1
  li      $t1, 0                    # int j = 0
main_forloop2:                      # for (j = 0; j < col size; j++)
  bge     $t1, 8, main_endforloop2
  mul     $t2, $t0, 8               # idx = col size * i
  add     $t2, $t2, $t1             # idx = col size * i + j
  lb      $t3, arr($t2)             # arr(idx) = input
main_if2:
  bne     $s0, 'W', main_else2      # if color == 'W':
  bne     $t3, 'b', main_elseif2_1  #   if tile == 'b':
  addi    $s2, $s2, 1               #   enemycount++
main_elseif2_1:
  bne     $t3, 'B', main_endif2     #   elif tile == 'B':
  addi    $s2, $s2, 1               #   enemycount++
  j       main_endif2
main_else2:                         # else:
  bne     $t3, 'w', main_elseif2_2  #   if tile == 'w':
  addi    $s2, $s2, 1               #   enemycount++
main_elseif2_2:
  bne     $t3, 'W', main_endif2     #   elif tile == 'W':
  addi    $s2, $s2, 1               #   enemycount++
main_endif2:
  addi    $t1, $t1, 1               # j++
  j       main_forloop2
main_endforloop2:                   # end for
  addi    $t0, $t0, 1               # i++
  j       main_forloop1
main_endforloop1:
  li      $v0, 0
```

Procedure 3

procedure MAINLOOP1($color, arr$)
    for $i \leftarrow 0$ to 7 **do**
        for $j \leftarrow 0$ to 7 **do**
            if $color =$ ‘w’ **then**
                if $arr[i][j] =$ ‘b’ or $arr[i][j] =$ ‘B’ **then**
                    $enemycount \leftarrow enemycount + 1$
            else
                if $arr[i][j] =$ ‘w’ or $arr[i][j] =$ ‘W’ **then**
                    $enemycount \leftarrow enemycount + 1$
    return $enemycount$

Table 1. Register usage

Register	Purpose
`$s0`	color of player to move
`$s1`	`'#'` character
`$s2`	temporary register
`$t0`	loop variable `i`
`$t1`	loop variable `j`
`$t2`	index `idx` of `arr`
`$t3`	element of `arr` at index `idx`
`$a0`	row parameter
`$a1`	column parameter
`$a2`	direction parameter
`$a3`	enemy count parameter

4 Checking for Solutions

After global variables are set, the array is traversed again and the function parameters for `jump` and `kjump` are set. The relevant labels are prefixed with `read`. The procedure is shown in Code Block 6 and desribed via pseudocode in Procedure 4. The use of registers is detailed in Table 1.

Procedure 4

1 **procedure** MAINLOOP2($color, arr, moves$)
2     $out \leftarrow 0$
3   **for** $i \leftarrow 0$ to 7 **do**
4   **for** $j \leftarrow 0$ to 7 **do**
5             $temp \leftarrow arr[i][j]$; $arr[i][j] \leftarrow$ '#' ▷ make current tile empty
6             $columnname \leftarrow$ 'a' $+ j$
7             $rowname \leftarrow 8 - i$
8             $moves[0] \leftarrow columnname + rowname +$ '␣' ▷ preemptively add first tile
9   **if** $color =$ 'W' **then**
10   **if** $arr[i][j] =$ 'w' **then** $out \leftarrow$ JUMP$(i, j, -1, enemycount)$
11   **else if** $arr[i][j] =$ 'W' **then** $out \leftarrow$ KJUMP$(i, j, -1, enemycount)$
12   **else**
13   **if** $arr[i][j] =$ 'b' **then** $out \leftarrow$ JUMP$(i, j, -1, enemycount)$
14   **else if** $arr[i][j] =$ 'B' **then** $out \leftarrow$ KJUMP$(i, j, -1, enemycount)$
15             $arr[i][j] \leftarrow temp$ ▷ restore tile
16   **if** $out = 1$ **then break** ▷ exit once a solution is found
17   **return** out

- Lines 1–12 and 44–48 are no different than the `for` loop in Code Block 4.
- Lines 14–15 temporarily empty the the current tile at index `idx`, while line 41 restores it. Like before, the current tile is stored at `$t3`.
- Lines 18–25 save the current tile name to `moves` regardless of whether a solution was found.
- Lines 26–40 facilitate function calls. The color `$s0` is first checked, then if the current tile is of the same color, `jump` or `kjump` is called, as in lines 10–14 of Procedure 4.
- Line 42 causes the loop to break when a solution is already found.

Code Block 6 Calling functions

```
read:
  li      $t0, 0                    # int i = 0
  li      $s1, '#'                  # load hash char
  li      $a2, -1                   # direction parameter
  move    $a3, $s2                  # enemycount parameter
read_forloop1:                      # for (i = 0; i < row size; i++)
  bge     $t0, 8, read_endforloop1
  li      $t1, 0                    # int j = 0
read_forloop2:                      # for (j = 0; j < col size; j++)
  bge     $t1, 8, read_endforloop2
  mul     $t2, $t0, 8               # idx = col size * i
  add     $t2, $t2, $t1             # idx = col size * i + j
#--------------------------------------------------------------------------------
  lb      $t3, arr($t2)             # get current tile at idx
  sb      $s1, arr($t2)             # make current tile empty
  move    $a0, $t0                  # row parameter
  move    $a1, $t1                  # column parameter
  li      $s2, 'a'                  # set column name
  add     $s2, $s2, $a1
  sb      $s2, moves
  li      $s2, '8'                  # set row name
  sub     $s2, $s2, $a0
  sb      $s2, moves+1
  li      $s2, '␣'
  sb      $s2, moves+2              # moves now contains "<tilename> "
read_if1:
  bne     $s0, 'W', read_else1      # if color == 'W':
  bne     $t3, 'w', read_elseif1_1  #   if tile == 'w':
  jal     jump                      #   call jump
read_elseif1_1:
  bne     $t3, 'W', read_endif1     #   elif tile == 'W':
  jal     kjump                     #   call kjump
  j       read_endif1
read_else1:                         # else:
  bne     $t3, 'b', read_elseif1_2  #   if tile == 'b':
  jal     jump                      #   call jump
read_elseif1_2:
  bne     $t3, 'B', read_endif1     #   elif tile == 'B':
  jal     kjump                     #   call kjump
read_endif1:
  sb      $t3, arr($t2)             # return tile to original state
  beq     $v0, 1, output            # break if solution found
#--------------------------------------------------------------------------------
  addi    $t1, $t1, 1               # j++
  j       read_forloop
read_endforloop2:                   # end for
  addi    $t0, $t0, 1               # i++
  j       read_forloop1
read_endforloop1:
```

Code Block 7 Printing the result

```
output:
  move    $s0, $v0            # print result
  li      $v0, 4
read_if2:
  beqz    $s0, read_else2     # if solution found: print "YES"
  la      $a0, yes
  syscall
  lw      $t1, cur            # print winning moves
  mul     $t2, $t1, 6         # add null character to end of string
  subi    $t2, $t2, 1         # to indicate end of moves
  sb      $0,  moves($t2)
  li      $v0, 4
  la      $a0, moves
  syscall
  j       read_endif2
read_else2:                   # else: print "NO"
  la      $a0, no
  syscall
read_endif2:
  li      $v0, 10
  syscall
```

Table 2. Direction key for `$a2`

Value	Direction
1	up and left
2	up and right
3	down and left
4	down and right
-1	initial jump (no direction)

Code Block 7 details how the output is handled. The output is moved from `$v0` to `$s0` since syscalls will be performed. Depending on the result, either the address of `yes` or `no` is loaded and then a string is printed. If a solution is found, the address of `moves` is loaded and the byte at the index `6 * cur`[1] is set to `\0` to ensure that any written but unused moves are not included in the output. Lastly, `moves` is printed and the program terminates.

5 The `jump` Function

The function `jump`, with its eponymous label prefixes, is responsible for checking solutions when a man of the same color as the player is encountered on the board. From Table 1, the function takes the row and column of the man's tile, the direction in which it jumps, and the current number of enemies. Table 2 details the corresponding directions to values of `$a2`. Procedure 5 describes in whole the algorithm that is adopted in the MIPS assembly code.

[1] Recall that a move is described in six characters, with the last character being a newline. Overwriting the character at the said index will then replace the newline at the last saved move.

Procedure 5 `jump` pseudocode

```
    Global: arr, color, enemies, kingrow, cur, moves
1   function Jump(row, column, direction, enemycount)
2       if direction = −1 then jump to line 23
3       if row, column ≤ −1 or row, column ≥ 8 then return 0                ▷ out of bounds
4       case
5           :direction = 1:  row′ ← row + 1;   column′ ← column + 1
6           :direction = 2:  row′ ← row + 1;   column′ ← column − 1
7           :direction = 3:  row′ ← row − 1;   column′ ← column + 1
8           :direction = 4:  row′ ← row − 1;   column′ ← column − 1
9           :direction = −1: pass
10      if arr[row′][col′] ∈ enemies and arr[row][col] = '#' then
11          enemycount ← enemycount − 1
12          cur        ← cur + 1                      ▷ arr[row′][col′] denotes jumped-over tile
13          columnname ← 'a' + j
14          rowname    ← 8 − i
15          newmove    ← columnname + rowname
16          k          ← (6 · cur) + 3
17          moves[k]   ← newmove + '\n' + newmove + '␣'                         ▷ write move
18          if enemycount = 0 then return 1                                      ▷ base case
19          else if row = kingrow then return 0
20          else                                                            ▷ recursive step
21              temp           ← arr[row′][col′];
22              arr[row′][col′] ← 'X'                                  ▷ remove captured tile
23              if color = 'W' then
24                  out ← [ Jump(row − 2, column − 2, 1, enemycount)
25                       or  Jump(row − 2, column + 2, 2, enemycount) ]
26              else
27                  out ← [ Jump(row + 2, column − 2, 3, enemycount)
28                       or  Jump(row + 2, column + 2, 4, enemycount) ]
29              if direction ≠ −1 then arr[row′][col′] ← temp                   ▷ restore tile
30              if out ≠ 1 then cur ← cur − 1
31              return out
32      else
33          return 0
```

Code Block 8 Setting up the stack frame

```
jump:
  ####preamble####
  subi    $sp, $sp, 32          # set up stack frame for 8 variables
  sw      $ra,  ($sp)
  sw      $a0, 4($sp)
  sw      $a1, 8($sp)
  sw      $a2, 12($sp)
  sw      $a3, 16($sp)
  sw      $s0, 20($sp)
  sw      $s1, 24($sp)
  sw      $s2, 28($sp)
  ####preamble####
```

5.1 Checking Parameters

As shown in Code Block 8, the function allocates stack space for 8 registers: a return address, four function parameters, and three `s` registers.

The first thing the function does is check whether it's the first time it has been called, as in Code Block 9. This is done by checking the value of `$a2`. If it equals -1, then the color of the piece is checked, and it jumps to the relevant labels (see Code Block 13) to begin recursing. This is analogous to Line 2 of Procedure 5.

Code Block 9 Checking initial call

```
  bne     $a2, -1, jump_endif1  # check if initial call:
  move    $s0, $a2              # use $s0 as temporary direction holder
  lb      $t4, color            # check color
jump_if1:
  bne     $t4, 'W', jump_else1  # if white:
  j       jump_if5
jump_else1:                     # else:
  j       jump_else5
jump_endif1:
```

To simplify code, the piece to move makes a jump (i.e., the function recurses) without checking whether the move is valid. The function then makes numerous checks after a jump to determine whether to return a value or to recurse.

The first check is confirming whether the piece is still inside the board after jumping. This is done by checking whether `$a0` and `$a1` are within the values 1–7, as in Code Block 10 and Line 3 of Procedure 5.

The function then checks whether the tile that the piece landed on is empty. If not, it returns 0, analogous to line 10 of the pseudocode.

Note that the first time the function is called, the direction is stored in `$s0`. This is because the tile restoration code in Line 32 of Code Block 13 is skipped the first time the function is called since no tile gets captured yet (see Line 29 of the pseudocode). Since `$a2` gets modified in the process of making recursive calls, its value must be saved in a separate register.

Code Block 10 Checking bounds and position

```
  li      $s1, 0                # set $s1 to 0 in case flag previously set to -1
jump_if2:                       # check if coordinates within bounds
  ble     $a0, -1, jump_return
  bge     $a0,  8, jump_return
  ble     $a1, -1, jump_return
  bge     $a1,  8, jump_return

  mul     $t4, $a0, 8           # idx = col size * row
  add     $t4, $t4, $a1         # idx = col size * row + col
  lb      $t5, arr($t4)         # get element at current tile

  bne     $t5, '#', jump_return # check if tile unoccupied
jump_endif2:
```

Code Block 11 Checking direction

```
jump_if3:                           # check direction
  bne     $a2, 1, jump_elseif3_1    # check if up, left
  addi    $s0, $a0, 1
  addi    $s1, $a1, 1
  j       jump_endif3
jump_elseif3_1:
  bne     $a2, 2, jump_elseif3_2    # check if up, right
  addi    $s0, $a0, 1
  addi    $s1, $a1, -1
  j       jump_endif3
jump_elseif3_2:
  bne     $a2, 3, jump_elseif3_3    # check if down, left
  addi    $s0, $a0, -1
  addi    $s1, $a1, 1
  j       jump_endif3
jump_elseif3_3:                     # check if down, right
  addi    $s0, $a0, -1
  addi    $s1, $a1, -1
jump_endif3:
```

Code Block 12 Capturing an enemy

```
  mul     $t4, $s0, 8           # idx = col size * row
  add     $t4, $t4, $s1         # idx = col size * row + col
  lb      $s2, arr($t4)         # get element at previous tile
#--------------------------------------------------------------------------------
jump_if4:                       # check if previous tile contained enemy
  lb      $t5, enemies
  beq     $s2, $t5, jump_else4
  lb      $t5, enemies + 1
  beq     $s2, $t5, jump_else4
  j       jump_return           # if no enemy, return 0
jump_else4:
  subi    $a3, $a3, 1           # decrement enemycount
#--------------------------------------------------------------------------------
  lw      $s1, cur              # load current move number
  addi    $s0, $s1, 1           # increment movecount
  sw      $s0, cur
  mul     $s1, $s1, 6
  addi    $s1, $s1, 3           # obtain current string index
#--------------------------------------------------------------------------------
  li      $s0, 'a'              # set column name
  add     $s0, $s0, $a1
  sb      $s0, moves($s1)       # modify both current and next line
  sb      $s0, moves+3($s1)
  li      $s0, '8'              # set row name
  sub     $s0, $s0, $a0
  sb      $s0, moves+1($s1)
  sb      $s0, moves+4($s1)
  li      $s0, '\n'
  sb      $s0, moves+2($s1)     # moves now includes "<this tile>\n<this tile> "
  li      $s0, '␣'
  sb      $s0, moves+5($s1)

  li      $s1, -1               # raise flag
jump_endif4:
  beq     $a3, 0, jump_BASE     # if no more enemies, return 1
  lw      $t5, kingrow          # else, if in enemy's king's row, return 0
  beq     $a0, $t5, jump_return
```

5.2 Capturing an Enemy

Line 10 of Procedure 5 checks whether the tile that was jumped over had an enemy and whether the current tile is empty. The latter condition was addressed in the previous section.

To check the jumped-over tile, its address is inferred from the direction parameter, as in Code Block 11 and Lines 4–9 of the pseudocode. Simply put, the function checks the value of `$a2` and stores the row and column of the tile one unit away in the opposite direction to `$s0` and `$s1`, respectively. From Lines 1–10 of Code Block 12, the calculated address is then stored in `$t4` and the jumped-over tile is stored in `$s2`. If this tile contains an enemy and the current tile is empty, it is captured and the enemy count is decremented. Otherwise, the function returns 0.

After capturing an enemy, the function saves the move to `moves` and performs a final check. If there are no more enemies, the function jumps to the base case (See Code Block 15) and returns 0. Else, if there are still enemies but the piece landed in the enemy's king row, it returns 0. Otherwise, it checks for more solutionchecks for more solutions.

5.3 Saving Moves

Lines 14–33 of Code Block 12 are responsible for saving which tiles were involved in a capture, which is equivalent to Lines 12–17 of Procedure 5. Recall from Section 2 that a move takes up six characters. Since the original tile is written preemptively, there is an offset of 3 characters. Thus, succeeding moves are written to `moves` at the index `(6 * cur) + 3` which is stored in `$s1`.

The current piece is written at the end of the last move and the beginning of the next move, as in Line 29 of the code block or Line 17 of the pseudocode. After the procedure is done, a flag is raised. This is done by storing the value `-1` is stored in `$s1`.

5.4 Recursive Step

Code Block 13 contains the code for the recursive step of the function. Before performing another jump, the tile that was successfully captured is temporarily replaced with an `'X'`, as in Line 2 of the code or Lines 21 and 22 of the pseudocode. This does nothing since men cannot jump backwards, but was kept anyway for bookkeeping.

Lines 7–28 perform the function of Lines 23–28 of the pseudocode. If the color is white, it recursively checks for solutions diagonally forward, and vice-versa. Lines 13, 17, and 24 make the function stop early when a solution is already found to save time. After recursing, the tile is restored to its original state and the function returns.

5.5 Keeping Track of Moves

In order to model the behavior of Line 30 of Procedure 5, a flag is raised to signal that an enemy was captured. From Section 5.3, this flag is `$s1` which holds the value `-1` when an enemy is captured. From Code Block 14, the `slti` instruction stores the value `1` in `$s1` if the flag was raised. Otherwise, it stores the value `0`. This value then gets subtracted from `cur` when no solution has been found. It's easy to see that this construction only decrements `cur` whenever the flag is raised.

5.6 Base Case

The base case of the function is quite simple. As can be seen from Code Block 15, the function emulates the `or` operations in Lines 24–25 and 27–28 of the pseudocode by only ever changing the value of `$v0` when a solution is found. Otherwise, `$v0` retains its original value which is zero.

Code Block 13 Recursive step

```
jump_recurse:                            # else, recurse
  li      $t5, 'X'                       # remove enemy from previous tile
  sb      $t5, arr($t4)
  move    $s0, $t4
  lb      $t4, color                     # check color
#--------------------------------------------------------------------------------
jump_if5:
  bne     $t4, 'W', jump_else5           # if white:
  subi    $a0, $a0, 2                    # check up, left
  subi    $a1, $a1, 2
  li      $a2, 1
  jal     jump
  beq     $v0, 1, jump_endrecurse        # break if solution found
  addi    $a1, $a1, 4                    # check up, right
  li      $a2, 2
  jal     jump
  beq     $v0, 1, jump_endrecurse
  j       jump_endif5
jump_else5:                              # else
  addi    $a0, $a0, 2                    # check down, left
  subi    $a1, $a1, 2
  li      $a2, 3
  jal     jump
  beq     $v0, 1, jump_endrecurse
  addi    $a1, $a1, 4                    # check down, right
  li      $a2, 4
  jal     jump
jump_endif5:
#--------------------------------------------------------------------------------
jump_endrecurse:
  beq     $s0, -1, jump_return           # check if initial call
  sb      $s2, arr($s0)                  # restore tile
  j       jump_return
```

Code Block 14 Decrementing `moves`

```
jump_return:
  beq     $v0, 1, jump_return_sub        # decrement movecount if no solution found
  slti    $s1, $s1, 0                    # AND enemy was captured

  lw      $s0, cur
  sub     $s0, $s0, $s1
  sw      $s0, cur
```

Code Block 15 Return value

```
jump_BASE:
  li      $v0, 1
  j       jump_return
```

6 The `kjump` Function

The function `kjump`, similar to the previous function with namesake label prefixes, checks for solutions when an ally king is encountered. It uses the same parameters and the same associated directions from Tables 1 and 2. Its algorithm is described completely in Procedure 6 for reference.

For the most part, `kjump` is just a slightly modified version of `jump`. The procedures for setting up a stack frame, checking parameters, modifying `$v0`, and writing a move, among others, are no different than the code in Code Blocks 8, 11, 14 and 15. Thus, only the distinct parts will be shown in code blocks.

6.1 Checking Parameters

Similar to `jump`, the function checks whether it was called for the first time, which can be seen in Code Block 16. It also performs the same bounds checking, and also saves `$a2` to `$s0` the first time it is called.

Unlike `jump`, however, it no longer checks the color of the piece before deciding which direction to recurse, since any king can go in any direction. Additionally, it no longer returns zero upon landing on an occupied tile, unless the occupant is an ally or a captured piece. This is shown in Line 17 of Code Block 17 and Line 14 of Procedure 6. Also note that after passing initial checks, the destination tile (which is either an enemy or empty tile) is stored in `$t6` for a later check.

6.2 Capturing an Enemy

The mechanism for checking the jumped-over tile is similar to that of `jump` and is detailed in Code Block 18. The value and address of the tile are again stored in `$s2` and `$t4`, respectively.

Whereas `jump` will return 0 if the jumped-over tile is not occupied by an enemy, `kjump` does one more check. If the jumped-over tile is empty, it recurses in the same direction as in Code Block 20 and Line 37 of Procedure 6. If it is instead occupied by an ally or captured piece, it returns zero.

On the other hand, if there is an enemy in the jumped-over tile, it checks whether the current tile is already occupied. Recall that the value is stored in `$t6`. If `$t6 != '#'`, the function returns 0. Otherwise, it captures the enemy and the move is saved.

The code for saving the move is exactly like Code Block 12 except that the function no longer checks whether the piece landed on the enemy's king row. Once a piece is captured, a flag is again raised by loading `-1` to `$s1`, and the move count is decremented in the same way as in Code Block 14.

If there are no more enemies, the function returns 1 just like `jump`. Otherwise, it recurses.

6.3 Recursive Step

The main recursive step involves checking for solutions in the four diagonal directions, as in Code Block 19 and Lines 28–31 of Procedure 6 where the values of `$a0` and `$a1` changed depending on the direction. As with `jump`, the captured piece is temporarily replaced with an `'X'` and then restored after recursing, unless the function is called for the first time.

The other recursive step is called when the jumped-over piece is empty and the current piece is not occupied by an ally or a captured piece. It's purpose is to simply recurse in the same direction. To do this, the code checks `$a2` and infers the address of the next tile, analogous to the `case` statement in the pseudocode.

Procedure 6 `kjump` pseudocode

```
   Global: arr, color, enemies, cur, moves
1  function KJUMP(row, column, direction, enemycount)
2      if direction = −1 then jump to line 28
3      if row, column ≤ −1 or row, column ≥ 8 then return 0
4      case
5          :direction = 1: row′ ← row + 1;   column′ ← column + 1
6                          row″ ← row − 1;   column″ ← column − 1
7          :direction = 2: row′ ← row + 1;   column′ ← column − 1
8                          row″ ← row − 1;   column″ ← column + 1
9          :direction = 3: row′ ← row − 1;   column′ ← column + 1
10                         row″ ← row + 1;   column″ ← column − 1
11         :direction = 4: row′ ← row − 1;   column′ ← column − 1
12                         row″ ← row + 1;   column″ ← column + 1
13         :direction = −1: pass
14     if arr[row][col] ∉ enemies and arr[row][col] ≠ '#' then return 0
15     else if arr[row′][col′] ∈ enemies then
16         if arr[row][col] ≠ '#' then return 0
17         enemycount ← enemycount − 1
18         cur        ← cur + 1
19         columnname ← 'a' + j
20         rowname    ← 8 − i
21         newmove    ← columnname + rowname
22         k          ← (6 · cur) + 3
23         moves[k]   ← newmove + '\n' + newmove + '␣'
24         if enemycount = 0 then return 1                      ▷ base case
25         else                                                 ▷ recursive step
26             temp          ← arr[row′][col′];
27             arr[row′][col′] ← 'X'
28             out           ← [ JUMP(row − 2, column − 2, 1, enemycount)
29                               or JUMP(row − 2, column + 2, 2, enemycount)
30                               or JUMP(row + 2, column − 2, 3, enemycount)
31                               or JUMP(row + 2, column + 2, 4, enemycount) ]
32             if direction ≠ −1 then arr[row′][col′] ← temp
33             if out ≠ 1 then cur ← cur − 1
34             return out
35     else                                  ▷ arr[row″][col″] denotes "next" tile
36         if arr[row′][col′] ≠ '#' then return 0
37         return KJUMP(row″, col″, direction, enemycount)
```

Code Block 16 Checking initial call

```
bne     $a2, -1, kjump_if1    # check if initial call:
move    $s0, $a2              # use $s0 as temporary direction holder
j       kjump_recurse1_sub
```

Code Block 17 Checking bounds and position

```
kjump_if1:
  li    $s1, 0                    # set $s1 to 0

  ble   $a0, -1, kjump_return     # check if coordinates within bounds
  bge   $a0, 8, kjump_return
  ble   $a1, -1, kjump_return
  bge   $a1, 8, kjump_return

  mul   $t4, $a0, 8               # idx = col size * row
  add   $t4, $t4, $a1             # idx = col size * row + col
  lb    $t5, arr($t4)             # get element at current tile

  lb    $s2, enemies              # check if current tile has enemy
  beq   $s2, $t5, kjump_endif1
  lb    $s2, enemies + 1
  beq   $s2, $t5, kjump_endif1
  bne   $t5, '#', kjump_return    # if current tile captured/has ally, return 0
kjump_endif1:                     # else, check previous tile
  move  $t6, $t5
```

Code Block 18 Checking for allies or captured pieces

```
kjump_if3:                          # check if previous tile contained enemy
  lb      $t5, enemies
  beq     $s2, $t5, kjump_elseif3
  lb      $t5, enemies + 1
  beq     $s2, $t5, kjump_elseif3
  bne     $s2, '#', kjump_return    # return 0 if previous tile captured/has ally
  j       kjump_recurse2            # if no enemy, keep checking same direction
kjump_elseif3:
  bne     $t6, '#', kjump_return    # return zero if current tile also occupied
kjump_else3:                        # else, capture enemy
  subi    $a3, $a3, 1               # decrement enemycount
```

Code Block 19 First recursive step

```
kjump_recurse1:                      # else, recurse
  li      $t5, 'X'                   # remove enemy from previous tile
  sb      $t5, arr($t4)
  move    $s0, $t4
#--------------------------------------------------------------------------------
kjump_recurse1_sub:
  subi    $a0, $a0, 2                # check up, left
  subi    $a1, $a1, 2
  li      $a2, 1
  jal     kjump
  beq     $v0, 1, kjump_endrecurse1 # break if solution found
  addi    $a1, $a1, 4                # check up, right
  li      $a2, 2
  jal     kjump
  beq     $v0, 1, kjump_endrecurse1
  addi    $a0, $a0, 4                # check down, right
  li      $a2, 4
  jal     kjump
  beq     $v0, 1, kjump_endrecurse1
  subi    $a1, $a1, 4                # check down, left
  li      $a2, 3
  jal     kjump
#--------------------------------------------------------------------------------
kjump_endrecurse1:
  beq     $s0, -1, kjump_return     # check if no replacement was made
  sb      $s2, arr($s0)              # restore tile
  j       kjump_return
```

Code Block 20 Second recursive step

```
kjump_recurse2:
kjump_if4:                            # check direction
  bne     $a2, 1, kjump_elseif4_1    # check up, left
  subi    $a0, $a0, 1
  subi    $a1, $a1, 1
  jal     kjump
  j       kjump_endif4
kjump_elseif4_1:
  bne     $a2, 2, kjump_elseif4_2    # check up, right
  subi    $a0, $a0, 1
  addi    $a1, $a1, 1
  jal     kjump
  j       kjump_endif4
kjump_elseif4_2:
  bne     $a2, 3, kjump_elseif4_3    # check down, left
  addi    $a0, $a0, 1
  subi    $a1, $a1, 1
  jal     kjump
  j       kjump_endif4
kjump_elseif4_3:                      # check down, right
  addi    $a0, $a0, 1
  addi    $a1, $a1, 1
  jal     kjump
  j       kjump_endif4
kjump_endif4:
  j       kjump_return
```