Assembly Chess

Bora Mollamustafaoglu, Xu Ji & Gun Pinyo

Designing the Emulator

Structs help us tokenize the data

```
struct data_processing {
  bool_t immediate;
  byte_t opcode;
  bool_t set_condition_codes;
  byte_t reg_n;
  byte_t reg_dest;
  uint16_t operand2;
};

struct multiply {
  bool_t accumulate;
  bool_t set_condition_codes;
  byte_t reg_dest;
  byte_t reg_n;
  byte_t reg_s;
  byte_t reg_m;
};

struct single_data_transfer {
  bool_t immediate;
  bool_t pre_index;
  bool_t up;
  bool_t load;
  byte_t reg_n;
  byte_t reg_d;
  uint16_t offset;
};

struct branch {
  uint32_t offset;
};
```

```
typedef union {
  struct data_processing *data_proc;
  struct multiply *mult;
  struct single_data_transfer *sdt;
  struct branch *branch;
} instr_t;

struct decoded_instr {
  enum instr_type type;
  instr_t *instr;
};
```

Currently decoded instruction stored as a union of structures

Designing the Assembler

Instruction

```
0 Program_loop:  MOVEQ   R0, #0x00000001        ; r1 is index counter

0 program_loop:  moveq   r0, #0x00000001        ; r1 is index counter
```

inst_entry

"mov"

"eq r0, #0x00000001"

label_entry

"program_loop"

0

Extensions to the Assembler & Coding Practice

```
get_gpio_start:
  STR     lr, [sp, #4]!

  ADR     r0, _g_gpio_start
  LDR     r0, [r0]

  LDR     pc, [sp], #-4

_g_gpio_start:
  DCD     0x20200000
```

- Support all 16 condition flags for all 26 mnemonics
- All 16 Data Processing opcodes supported
- Barrel register shift supported
- ASCII immediates supported (for code readability)
- Branch with Link supported (allowing method calling)

- Write-back option on Single Data Transfer supported (allowing stack operations)
- DCD and ADR directives supported (allowing global variables and data structures)

```
BL  get_mailbox_base
```

Testsuite

Let’s play some chess!

Game Loop

- GPIO pins are set up to receive input
- The chess board is prepared

- Display board, get input, and process the input
- Keep looping until someone wins
- Restart the game

```
.main:
  main_while_all_game:
    BL init_stack
    BL init_pins
    BL initialise

    main_while_each_game:
      BL get_cur_player
      BL is_game_over
      CMP r0, #0
      BNE main_end_each_game

      BL display
      BL manage_input
      BL get_is_clicked
      CMP r0, #0
      BLNE process
      B main_while_each_game

    main_end_each_game:
      BL display
      BL game_over

    B main_while_all_game
```

- Stack pointer (SP register) points to the start of a 1KB block of memory

```
init_stack:
    ADR     sp, stack
    MOV     pc, lr
stack:
    DCD     0
    DCD     0
    DCD     0
    DCD     0
    DCD     0
    DCD     0
    DCD     0
    DCD     0
    DCD     0
```

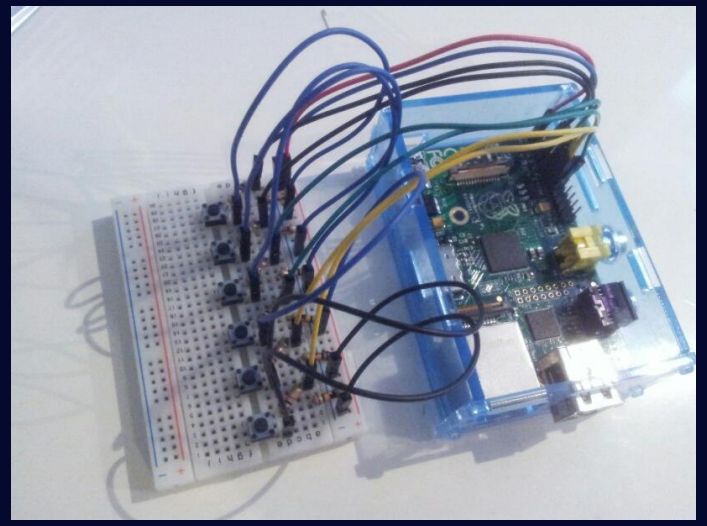
Input: Getting Data from the GPIO Pins

```
manage_input:
    STR     lr, [sp, #4]!
    STR     r0, [sp, #4]!
    STR     r1, [sp, #4]!
    STR     r2, [sp, #4]!

    BL      get_current_pos
    BL      reverse_row
    MOV     r1, r0              ; r1 holds current_pos
    MOV     r2, r0              ; r2 is another copy, to test if it changes

  manage_input_left:
    MOV     r0, #0xB6000        ; wait to stop cursor from moving
    BL      wait                ;  too quickly

    MOV     r0, #18
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_up

    MOV     r0, #0              ; not a click
    BL      set_is_clicked
    AND     r0, r1, #7          ; r0 = r1 % 8
    CMP     r0, #0
    SUBNE   r1, r1, #1          ; move cursor left
    ADDEQ   r1, r1, #7          ; wrap around
    B       manage_input_end

        ...

 manage_input_selected:
    MOV     r0, #22
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_reset

    MOV     r0, #1
    BL      set_is_clicked
    B       manage_input_end

 manage_input_reset:
    MOV     r0, #27
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_left
    B     .main

  manage_input_end:
    MOV     r0, r1              ; set current pos
    BL      reverse_row
    CMP     r1, r2              ; check if current_pos has been updated
    BLNE    set_current_pos
```

- *manage_input()* loops until it detects the next selected position from the player
- We update the display with every button clicked, but we only exit the loop when the player hits the “select”button
- Helper functions:
  - *wait()* delays the detection of input
  - *get_gpio_input(pin_number)* to read from a pin
  - *reverse_row(pos)* to turn the game state as stored in the pieces array into the game state as it should be displayed on screen

Buttons:

- left = GPIO18
- up = GPIO23
- down = GPIO4
- right = GPIO17
- select = GPIO22
- reset game = GPIO27



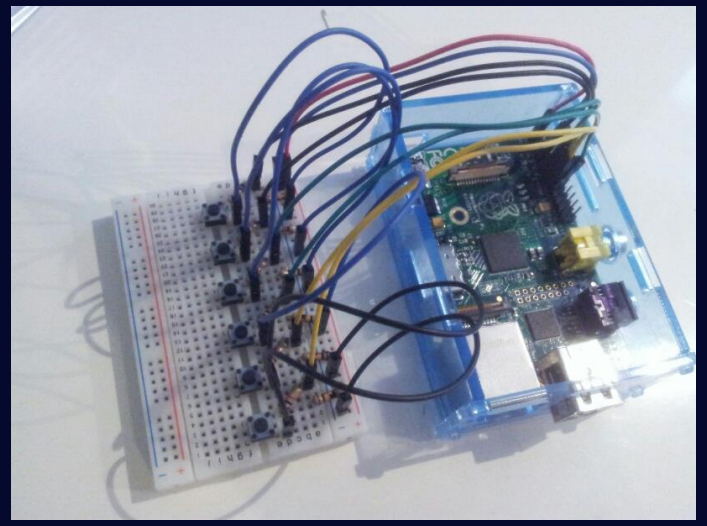
Input: Getting Data from the GPIO Pins

```
manage_input:
    STR     lr, [sp, #4]!
    STR     r0, [sp, #4]!
    STR     r1, [sp, #4]!
    STR     r2, [sp, #4]!

    BL      get_current_pos
    BL      reverse_row
    MOV     r1, r0              ; r1 holds current_pos
    MOV     r2, r0              ; r2 is another copy, to test if it changes

  manage_input_left:
    MOV     r0, #0xB6000        ; wait to stop cursor from moving
    BL      wait                ;  too quickly

    MOV     r0, #18
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_up

    MOV     r0, #0              ; not a click
    BL      set_is_clicked
    AND     r0, r1, #7          ; r0 = r1 % 8
    CMP     r0, #0
    SUBNE   r1, r1, #1          ; move cursor left
    ADDEQ   r1, r1, #7          ; wrap around
    B       manage_input_end

        ...

 manage_input_selected:
    MOV     r0, #22
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_reset

    MOV     r0, #1
    BL      set_is_clicked
    B       manage_input_end

 manage_input_reset:
    MOV     r0, #27
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_left
    B     .main

  manage_input_end:
    MOV     r0, r1              ; set current pos
    BL      reverse_row
    CMP     r1, r2              ; check if current_pos has been updated
    BLNE    set_current_pos
```

- *manage_input()* loops until it detects the next selected position from the player
- We update the display with every button clicked, but we only exit the loop when the player hits the “select”button
- Helper functions:
  - *wait()* delays the detection of input
  - *get_gpio_input(pin_number)* to read from a pin
  - *reverse_row(pos)* to turn the game state as stored in the pieces array into the game state as it should be displayed on screen

Buttons:

- left = GPIO18
- up = GPIO23
- down = GPIO4
- right = GPIO17
- select = GPIO22
- reset game = GPIO27



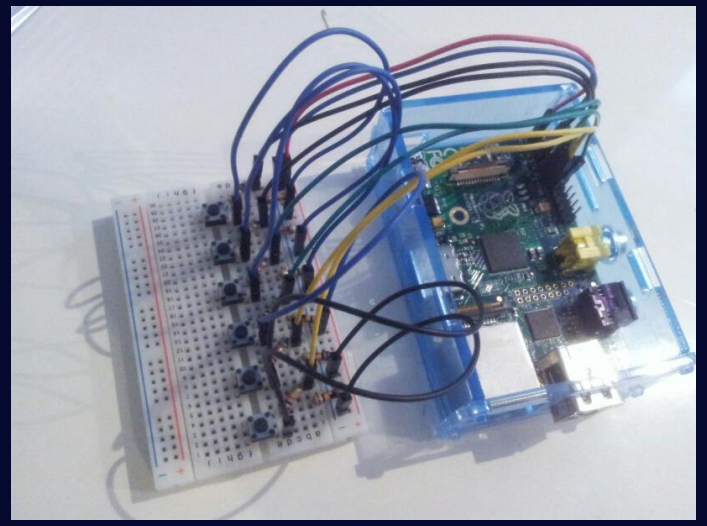
Input: Getting Data from the GPIO Pins

```
manage_input:
    STR     lr, [sp, #4]!
    STR     r0, [sp, #4]!
    STR     r1, [sp, #4]!
    STR     r2, [sp, #4]!

    BL      get_current_pos
    BL      reverse_row
    MOV     r1, r0              ; r1 holds current_pos
    MOV     r2, r0              ; r2 is another copy, to test if it changes

  manage_input_left:
    MOV     r0, #0xB6000        ; wait to stop cursor from moving
    BL      wait                ;  too quickly

    MOV     r0, #18
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_up

    MOV     r0, #0              ; not a click
    BL      set_is_clicked
    AND     r0, r1, #7          ; r0 = r1 % 8
    CMP     r0, #0
    SUBNE   r1, r1, #1          ; move cursor left
    ADDEQ   r1, r1, #7          ; wrap around
    B       manage_input_end

        ...

 manage_input_selected:
    MOV     r0, #22
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_reset

    MOV     r0, #1
    BL      set_is_clicked
    B       manage_input_end

 manage_input_reset:
    MOV     r0, #27
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_left
    B     .main

  manage_input_end:
    MOV     r0, r1              ; set current pos
    BL      reverse_row
    CMP     r1, r2              ; check if current_pos has been updated
    BLNE    set_current_pos
```

- *manage_input()* loops until it detects the next selected position from the player
- We update the display with every button clicked, but we only exit the loop when the player hits the “select”button
- Helper functions:
  - *wait()* delays the detection of input
  - *get_gpio_input(pin_number)* to read from a pin
  - *reverse_row(pos)* to turn the game state as stored in the pieces array into the game state as it should be displayed on screen

Buttons:

- left = GPIO18
- up = GPIO23
- down = GPIO4
- right = GPIO17
- select = GPIO22
- reset game = GPIO27



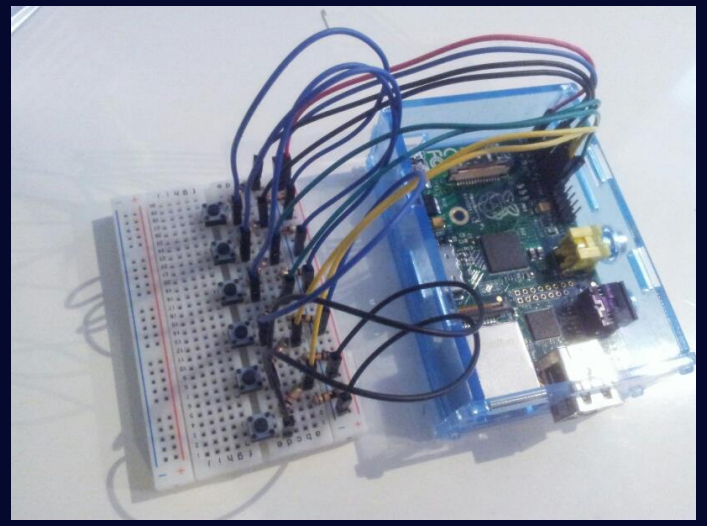
Input: Getting Data from the GPIO Pins

```
manage_input:
    STR     lr, [sp, #4]!
    STR     r0, [sp, #4]!
    STR     r1, [sp, #4]!
    STR     r2, [sp, #4]!

    BL      get_current_pos
    BL      reverse_row
    MOV     r1, r0              ; r1 holds current_pos
    MOV     r2, r0              ; r2 is another copy, to test if it changes

  manage_input_left:
    MOV     r0, #0xB6000        ; wait to stop cursor from moving
    BL      wait                ;  too quickly

    MOV     r0, #18
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_up

    MOV     r0, #0              ; not a click
    BL      set_is_clicked
    AND     r0, r1, #7          ; r0 = r1 % 8
    CMP     r0, #0
    SUBNE   r1, r1, #1          ; move cursor left
    ADDEQ   r1, r1, #7          ; wrap around
    B       manage_input_end

        ...

 manage_input_selected:
    MOV     r0, #22
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_reset

    MOV     r0, #1
    BL      set_is_clicked
    B       manage_input_end

 manage_input_reset:
    MOV     r0, #27
    BL      get_gpio_input
    CMP     r0, #0
    BNE     manage_input_left
    B     .main

  manage_input_end:
    MOV     r0, r1              ; set current pos
    BL      reverse_row
    CMP     r1, r2              ; check if current_pos has been updated
    BLNE    set_current_pos
```

- *manage_input()* loops until it detects the next selected position from the player
- We update the display with every button clicked, but we only exit the loop when the player hits the “select”button
- Helper functions:
  - *wait()* delays the detection of input
  - *get_gpio_input(pin_number)* to read from a pin
  - *reverse_row(pos)* to turn the game state as stored in the pieces array into the game state as it should be displayed on screen

Buttons:

- left = GPIO18
- up = GPIO23
- down = GPIO4
- right = GPIO17
- select = GPIO22
- reset game = GPIO27



Output: Displaying the Chessboard

```
display_loop:
    CMP     r9, #64
    BGE     display_end                     ; *set function parameters*
        CMP     r10, #8                     ; if end of row reached, update row ind + addr
        MOVEQ   r10, #0
        ADDEQ   r5, r5, r6
        SUBEQ   r11, r8, r11                ; invert colour

        MOV     r0, r9                      ; r3 = piece colour, r4 = outline colour
        BL      reverse_row
        BL      get_cells_side
        BL      display_get_colour          ; returns piece colour and outline colour
        MOV     r3, r0
        MOV     r4, r1
        MOV     r0, r9                      ; r1 = piece type
        BL      reverse_row
        BL      get_cells_type
        MOV     r1, r0
        MOV     r2, r11                     ; r2 = background colour
        STR     r5, [sp, #4]!
        BL      get_length_square
        MOV     r12, r0
        MOV     r0, r5                      ; r0 = current address
        MOV     r5, r12
        BL      draw_square
        LDR     r5, [sp], #-4               ; ******************************************

        MOV     r0, r9
        BL      reverse_row
        BL      get_are_marked              ; order matters here
        CMP     r0, #0
        BLNE    get_are_marked_colour
        MOVNE   r1, r0
        MOVNE   r0, r5
        BLNE    draw_border                 ; ******************************************

        BL      get_has_selected
        CMP     r0, #0
        BEQ     display_skip_selected
        BL      get_selected_pos
        BL      reverse_row
        CMP     r0, r9
        BLEQ    get_selected_pos_colour
        MOVEQ   r1, r0
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; ******************************************

    display_skip_selected:
        BL      get_cur_player
        CMP     r0, #0                      ; check whether white/black
        BLEQ    get_player_one_colour       ; set up cursor depending
        BLNE    get_player_two_colour       ; current player
        MOV     r1, r0
        BL      get_current_pos             ; show border if correct
        BL      reverse_row
        CMP     r0, r9                      ; square
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; ******************************************
```

- *display()* loops through the 64 fields of the *cells_type*, *cells_side*, *are_marked* arrays to render the chessboard
- It writes each pixel to the GPU framebuffer, but this detail is abstracted away in its helper functions
- Helper functions
  - *get_length_square()* returns the length of a single square. Our code is not board-specific – we could use squares of any size
  - *draw_square(address, piece type, background colour, piece colour, piece outline colour)* renders a single square
  - *draw_border(address, colour)* renders the border around a single square

Output: Displaying the Chessboard

```
display_loop:
    CMP     r9, #64
    BGE     display_end                     ; *set function parameters*
        CMP     r10, #8                     ; if end of row reached, update row ind + addr
        MOVEQ   r10, #0
        ADDEQ   r5, r5, r6
        SUBEQ   r11, r8, r11                ; invert colour

        MOV     r0, r9                      ; r3 = piece colour, r4 = outline colour
        BL      reverse_row
        BL      get_cells_side
        BL      display_get_colour          ; returns piece colour and outline colour
        MOV     r3, r0
        MOV     r4, r1
        MOV     r0, r9                      ; r1 = piece type
        BL      reverse_row
        BL      get_cells_type
        MOV     r1, r0
        MOV     r2, r11                     ; r2 = background colour
        STR     r5, [sp, #4]!
        BL      get_length_square
        MOV     r12, r0
        MOV     r0, r5                      ; r0 = current address
        MOV     r5, r12
        BL      draw_square
        LDR     r5, [sp], #-4               ; ******************************************

        MOV     r0, r9
        BL      reverse_row
        BL      get_are_marked              ; order matters here
        CMP     r0, #0
        BLNE    get_are_marked_colour
        MOVNE   r1, r0
        MOVNE   r0, r5
        BLNE    draw_border                 ; ******************************************

        BL      get_has_selected
        CMP     r0, #0
        BEQ     display_skip_selected
        BL      get_selected_pos
        BL      reverse_row
        CMP     r0, r9
        BLEQ    get_selected_pos_colour
        MOVEQ   r1, r0
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; ******************************************

    display_skip_selected:
        BL      get_cur_player
        CMP     r0, #0                      ; check whether white/black
        BLEQ    get_player_one_colour       ; set up cursor depending
        BLNE    get_player_two_colour       ; current player
        MOV     r1, r0
        BL      get_current_pos             ; show border if correct
        BL      reverse_row
        CMP     r0, r9                      ; square
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; ******************************************
```

- *display()* loops through the 64 fields of the *cells_type*, *cells_side*, *are_marked* arrays to render the chessboard
- It writes each pixel to the GPU framebuffer, but this detail is abstracted away in its helper functions
- Helper functions
  - *get_length_square()* returns the length of a single square. Our code is not board-specific – we could use squares of any size
  - *draw_square(address, piece type, background colour, piece colour, piece outline colour)* renders a single square
  - *draw_border(address, colour)* renders the border around a single square

Output: Displaying the Chessboard

```
display_loop:
    CMP     r9, #64
    BGE     display_end                     ; *set function parameters*
        CMP     r10, #8                     ; if end of row reached, update row ind + addr
        MOVEQ   r10, #0
        ADDEQ   r5, r5, r6
        SUBEQ   r11, r8, r11                ; invert colour

        MOV     r0, r9                      ; r3 = piece colour, r4 = outline colour
        BL      reverse_row
        BL      get_cells_side
        BL      display_get_colour          ; returns piece colour and outline colour
        MOV     r3, r0
        MOV     r4, r1
        MOV     r0, r9                      ; r1 = piece type
        BL      reverse_row
        BL      get_cells_type
        MOV     r1, r0
        MOV     r2, r11                     ; r2 = background colour
        STR     r5, [sp, #4]!
        BL      get_length_square
        MOV     r12, r0
        MOV     r0, r5                      ; r0 = current address
        MOV     r5, r12
        BL      draw_square
        LDR     r5, [sp], #-4               ; ****************************************

        MOV     r0, r9
        BL      reverse_row
        BL      get_are_marked              ; order matters here
        CMP     r0, #0
        BLNE    get_are_marked_colour
        MOVNE   r1, r0
        MOVNE   r0, r5
        BLNE    draw_border                 ; *******************************************

        BL      get_has_selected
        CMP     r0, #0
        BEQ     display_skip_selected
        BL      get_selected_pos
        BL      reverse_row
        CMP     r0, r9
        BLEQ    get_selected_pos_colour
        MOVEQ   r1, r0
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; *******************************************

    display_skip_selected:
        BL      get_cur_player
        CMP     r0, #0                      ; check whether white/black
        BLEQ    get_player_one_colour       ; set up cursor depending
        BLNE    get_player_two_colour       ; current player
        MOV     r1, r0
        BL      get_current_pos             ; show border if correct
        BL      reverse_row
        CMP     r0, r9                      ; square
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; *******************************************
```

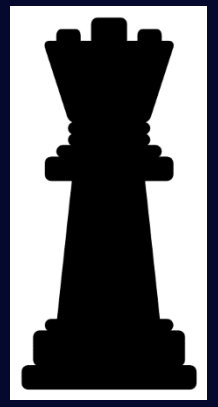
- *display()* loops through the 64 fields of the *cells_type*, *cells_side*, *are_marked* arrays to render the chessboard
- It writes each pixel to the GPU framebuffer, but this detail is abstracted away in its helper functions
- Helper functions
  - *get_length_square()* returns the length of a single square. Our code is not board-specific – we could use squares of any size
  - *draw_square(address, piece type, background colour, piece colour, piece outline colour)* renders a single square
  - *draw_border(address, colour)* renders the border around a single square

Output: Displaying the Chessboard

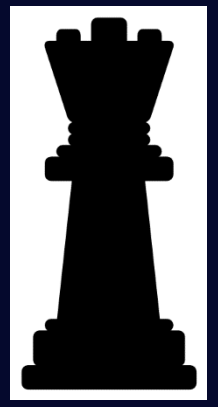
```
display_loop:
    CMP     r9, #64
    BGE     display_end                     ; *set function parameters*
        CMP     r10, #8                     ; if end of row reached, update row ind + addr
        MOVEQ   r10, #0
        ADDEQ   r5, r5, r6
        SUBEQ   r11, r8, r11                ; invert colour

        MOV     r0, r9                      ; r3 = piece colour, r4 = outline colour
        BL      reverse_row
        BL      get_cells_side
        BL      display_get_colour          ; returns piece colour and outline colour
        MOV     r3, r0
        MOV     r4, r1
        MOV     r0, r9                      ; r1 = piece type
        BL      reverse_row
        BL      get_cells_type
        MOV     r1, r0
        MOV     r2, r11                     ; r2 = background colour
        STR     r5, [sp, #4]!
        BL      get_length_square
        MOV     r12, r0
        MOV     r0, r5                      ; r0 = current address
        MOV     r5, r12
        BL      draw_square
        LDR     r5, [sp], #-4               ; ****************************************

        MOV     r0, r9
        BL      reverse_row
        BL      get_are_marked              ; order matters here
        CMP     r0, #0
        BLNE    get_are_marked_colour
        MOVNE   r1, r0
        MOVNE   r0, r5
        BLNE    draw_border                 ; ********************************************

        BL      get_has_selected
        CMP     r0, #0
        BEQ     display_skip_selected
        BL      get_selected_pos
        BL      reverse_row
        CMP     r0, r9
        BLEQ    get_selected_pos_colour
        MOVEQ   r1, r0
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; ********************************************

    display_skip_selected:
        BL      get_cur_player
        CMP     r0, #0                      ; check whether white/black
        BLEQ    get_player_one_colour       ; set up cursor depending
        BLNE    get_player_two_colour       ; current player
        MOV     r1, r0
        BL      get_current_pos             ; show border if correct
        BL      reverse_row
        CMP     r0, r9                      ; square
        MOVEQ   r0, r5
        BLEQ    draw_border                 ; ********************************************
```

- *display()* loops through the 64 fields of the *cells_type*, *cells_side*, *are_marked* arrays to render the chessboard
- It writes each pixel to the GPU framebuffer, but this detail is abstracted away in its helper functions
- Helper functions
  - *get_length_square()* returns the length of a single square. Our code is not board-specific – we could use squares of any size
  - *draw_square(address, piece type, background colour, piece colour, piece outline colour)* renders a single square
  - *draw_border(address, colour)* renders the border around a single square

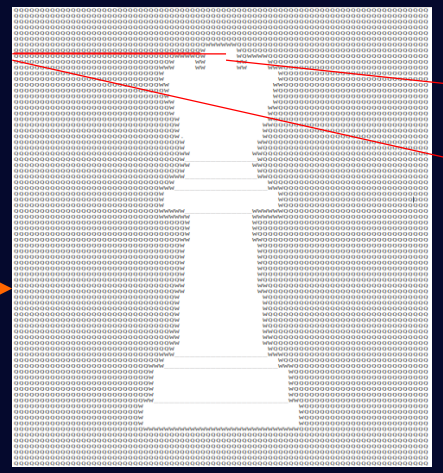
Graphics



Graphics

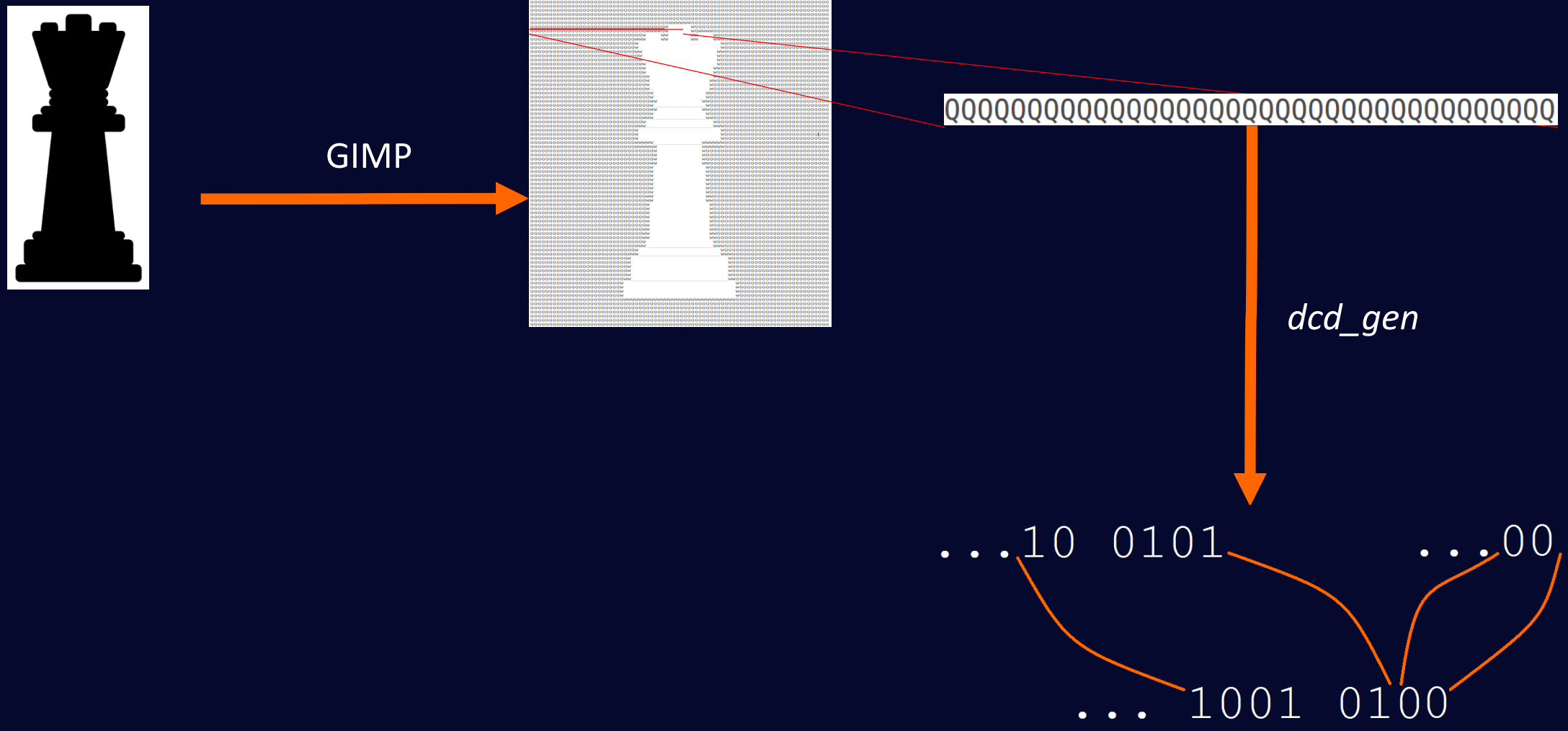


GIMP

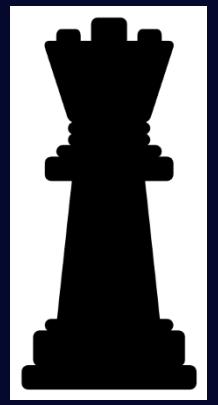


QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ

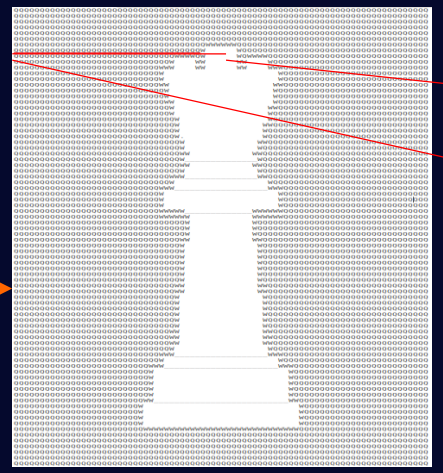
Graphics


GIMP
QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ
dcd_gen
...10 0101
...00
... 1001 0100

Graphics



GIMP



QQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQQ

dcd_gen

...10 0101 ...00

... 1001 0100

dcd_gen

```
get_queen_array:
    ADR     r0, _g_queen_array
    MOV     pc, lr
_g_queen_array:
    DCD     267
    DCD     2068
    DCD     26
    DCD     292
    DCD     6
    DCD     25
    DCD     6
    DCD     268
    DCD     18
    DCD     4
    DCD     6
    DCD     25
    DCD     6
    DCD     4
    DCD     18
    DCD     244
    DCD     6
    DCD     17
    DCD     10
    DCD     25
    DCD     10
    DCD     17
    DCD     6
    DCD     232
    DCD     14
    DCD     17
    DCD     10
    DCD     25
    DCD     10
```

Game Logic – Global Variables

- *cur_player* — Whose turn is it? 0 for first player, 1 for second player
- *has_selected* — Has the the current player selected their piece yet?
- *is_clicked* — Has this iteration seen a clicked event?

Game Logic – Global Variables

- *cur_player* — Whose turn is it? 0 for first player, 1 for second player
- *has_selected* — Has the the current player selected their piece yet?
- *is_clicked* — Has this iteration seen a clicked event?

- *en_passant_flag* — Status that is associated with en passant in legal move
- *castle_flag* — Status that is associated with castle in legal move

Game Logic – Global Variables

- *cur_player* — Whose turn is it? 0 for first player, 1 for second player
- *has_selected* — Has the the current player selected their piece yet?
- *is_clicked* — Has this iteration seen a clicked event?

- *en_passant_flag* — Status that is associated with en passant in legal move
- *castle_flag* — Status that is associated with castle in legal move

- *selected_pos* — Position of the piece that current player selected (valid iff has_selected == true)
- *current_pos* — Position of cursor

Game Logic – Global Variables

- *cur_player* — Whose turn is it? 0 for first player, 1 for second player
- *has_selected* — Has the the current player selected their piece yet?
- *is_clicked* — Has this iteration seen a clicked event?

- *en_passant_flag* — Status that is associated with en passant in legal move
- *castle_flag* — Status that is associated with castle in legal move

- *selected_pos* — Position of the piece that current player selected (valid iff has_selected == true)
- *current_pos* — Position of cursor

For each cell *i* on the board, *i* ∈ [0, 64):

- *cells_type[i]* — Type of this cell: 'S' 'P' 'H' 'B' 'R' 'Q' 'K'
- *are_marked[i]* — Can the piece on selected_pos move to this cell legally?
- *cells_side[i]* — Which player does this cell belong to?

Game Logic – process()

```
void process() {
  assert(0 <= current_pos && current_pos < BOARD_SIZE);
  if(!has_selected) {
    if(!is_own_piece(current_pos)) // illegal selection
      return;
    for(byte_t iter = 0; iter<BOARD_SIZE; iter++)
      are_marked[iter] = !is_in_check(cur_player,current_pos,iter)
                         && legal_move(current_pos,iter,FALSE);
    selected_pos = current_pos;
    has_selected = TRUE;
  } else {
    assert(0 <= selected_pos && selected_pos < BOARD_SIZE);
    if(are_marked[current_pos]) { // click on legal cell
      legal_move(selected_pos,current_pos,TRUE);
      cur_player = !cur_player;
    }
    for(byte_t iter = 0; iter<BOARD_SIZE; iter++)
      are_marked[iter] = FALSE;
    selected_pos = BYTE_UNDEFINED;
    has_selected = FALSE;
  }
}
```

Game Logic – process()

```
MOV     r2, r8
BL      get_current_pos
MOV     r1, r0
BL      get_cur_player
BL      is_in_check
RSB     r3, r0, #1

BL      get_current_pos
MOV     r1, r8
MOV     r2, #0
BL      legal_move

AND     r1, r0, r3
MOV     r0, r8
BL      set_are_marked
```

```
void process() {
  assert(0 <= current_pos && current_pos < BOARD_SIZE);
  if(!has_selected) {
    if(!is_own_piece(current_pos)) // illegal selection
      return;
    for(byte_t iter = 0; iter<BOARD_SIZE; iter++)
      are_marked[iter] = !is_in_check(cur_player,current_pos,iter)
                         && legal_move(current_pos,iter,FALSE);
    selected_pos = current_pos;
    has_selected = TRUE;
  } else {
    assert(0 <= selected_pos && selected_pos < BOARD_SIZE);
    if(are_marked[current_pos]) { // click on legal cell
      legal_move(selected_pos,current_pos,TRUE);
      cur_player = !cur_player;
    }
    for(byte_t iter = 0; iter<BOARD_SIZE; iter++)
      are_marked[iter] = FALSE;
    selected_pos = BYTE_UNDEFINED;
    has_selected = FALSE;
  }
}
```

Game Logic – process()

```
MOV     r2, r8
BL      get_current_pos
MOV     r1, r0
BL      get_cur_player
BL      is_in_check
RSB     r3, r0, #1

BL      get_current_pos
MOV     r1, r8
MOV     r2, #0
BL      legal_move

AND     r1, r0, r3
MOV     r0, r8
BL      set_are_marked
```

```
process_else_has_selected_if_begin:
  MOV     r2, #1
  BL      get_current_pos
  MOV     r1, r0
  BL      get_selected_pos
  BL      legal_move
  BL      get_cur_player
  RSB     r0, r0, #1
  BL      set_cur_player
process_else_has_selected_if_end:
```

```
void process() {
  assert(0 <= current_pos && current_pos < BOARD_SIZE);
  if(!has_selected) {
    if(!is_own_piece(current_pos)) // illegal selection
      return;
    for(byte_t iter = 0; iter<BOARD_SIZE; iter++)
      are_marked[iter] = !is_in_check(cur_player,current_pos,iter)
                         && legal_move(current_pos,iter,FALSE);
    selected_pos = current_pos;
    has_selected = TRUE;
  } else {
    assert(0 <= selected_pos && selected_pos < BOARD_SIZE);
    if(are_marked[current_pos]) { // click on legal cell
      legal_move(selected_pos,current_pos,TRUE);
      cur_player = !cur_player;
    }
    for(byte_t iter = 0; iter<BOARD_SIZE; iter++)
      are_marked[iter] = FALSE;
    selected_pos = BYTE_UNDEFINED;
    has_selected = FALSE;
  }
}
```

Game Logic – is_in_check()

```
bool_t is_in_check(bool_t player_id, byte_t src_pos, byte_t des_pos) {
  byte_t tmp_cell_type = cells_type[des_pos];
  bool_t tmp_cell_side = cells_side[des_pos];
  bool_t tmp_cur_player = cur_player;
  cur_player = !player_id;
  byte_t king_pos,killer_pos;
  bool_t check = FALSE;
  actual_move(src_pos,des_pos);
  for(king_pos = 0; king_pos<BOARD_SIZE; king_pos++)
    if(cells_type[king_pos] == PIECE_K && cells_side[king_pos] == player_id)
      break;
  if(king_pos < BOARD_SIZE)
    for(killer_pos = 0; killer_pos<BOARD_SIZE; killer_pos++)
      check |= legal_move(killer_pos,king_pos,FALSE);
  actual_move(des_pos,src_pos);
  cells_type[des_pos] = tmp_cell_type;
  cells_side[des_pos] = tmp_cell_side;
  cur_player = tmp_cur_player;
  return check;
}
```

Game Logic – is_in_check()

```
is_in_check_king_begin:
  MOV     r6, #0
is_in_check_king_next:
  CMP     r6, #64
  BGE     is_in_check_return

  MOV     r0, r6
  BL      get_cells_type
  MOV     r1, r0
  MOV     r0, r6
  BL      get_cells_side
  CMP     r0, r10
  CMPEQ   r1, #'K'
  BEQ     is_in_check_king_end

  ADD     r6, r6, #1
  B       is_in_check_king_next
is_in_check_king_end:

is_in_check_killer_begin:
  MOV     r7, #0
is_in_check_killer_next:
  CMP     r7, #64
  BGE     is_in_check_killer_end

  MOV     r0, r7
  MOV     r1, r6
  MOV     r2, #0
  BL      legal_move
  ORR     r5, r5, r0

  ADD     r7, r7, #1
  B       is_in_check_killer_next
is_in_check_killer_end:
```

```
bool_t is_in_check(bool_t player_id, byte_t src_pos, byte_t des_pos) {
  byte_t tmp_cell_type = cells_type[des_pos];
  bool_t tmp_cell_side = cells_side[des_pos];
  bool_t tmp_cur_player = cur_player;
  cur_player = !player_id;
  byte_t king_pos,killer_pos;
  bool_t check = FALSE;
  actual_move(src_pos,des_pos);
  for(king_pos = 0; king_pos<BOARD_SIZE; king_pos++)
    if(cells_type[king_pos] == PIECE_K && cells_side[king_pos] == player_id)
      break;
  if(king_pos < BOARD_SIZE)
    for(killer_pos = 0; killer_pos<BOARD_SIZE; killer_pos++)
      check |= legal_move(killer_pos,king_pos,FALSE);
  actual_move(des_pos,src_pos);
  cells_type[des_pos] = tmp_cell_type;
  cells_side[des_pos] = tmp_cell_side;
  cur_player = tmp_cur_player;
  return check;
}
```

Game Logic – legal_move()

```
legal_move:
  STR     lr, [sp, #4]!              ; push lr
  BL      push_all_r0_r12
  MOV     r11, r0                    ; r11 as src_pos
  MOV     r12, r1                    ; r12 as des_pos
  MOV     r10, r2                    ; r10 as update

  MOV     r0, r11
  BL      is_own_piece
  CMP     r0, #0                     ; if(!is_own_piece(src_pos))
  BEQ     legal_move_return_false

  MOV     r0, r12
  BL      is_own_piece
  CMP     r0, #0                     ; if(is_own_piece(des_pos))
  BNE     legal_move_return_false

  CMP     r11, r12                   ; if(src_pos == des_pos)
  BEQ     legal_move_return_false

  MOV     r6, r11, LSR #3            ; r6 as src_row
  AND     r7, r11, #7                ; r7 as src_col
  MOV     r8, r12, LSR #3            ; r8 as des_row
  AND     r9, r12, #7                ; r8 as des_col

  SUB     r0, r6, r8
  BL      absolute
  MOV     r4, r0                    ; r4 as absolute(src_row - des_row), diff_row
  SUB     r0, r7, r9
  BL      absolute
  MOV     r5, r0                    ; r5 as absolute(src_col - des_col), diff_col
```

Game Logic – legal_move()

```
legal_move:
  STR     lr, [sp, #4]!              ; push lr
  BL      push_all_r0_r12
  MOV     r11, r0                    ; r11 as src_pos
  MOV     r12, r1                    ; r12 as des_pos
  MOV     r10, r2                    ; r10 as update

  MOV     r0, r11
  BL      is_own_piece
  CMP     r0, #0                     ; if(!is_own_piece(src_pos))
  BEQ     legal_move_return_false

  MOV     r0, r12
  BL      is_own_piece
  CMP     r0, #0                     ; if(is_own_piece(des_pos))
  BNE     legal_move_return_false

  CMP     r11, r12                   ; if(src_pos == des_pos)
  BEQ     legal_move_return_false

  MOV     r6, r11, LSR #3            ; r6 as src_row
  AND     r7, r11, #7                ; r7 as src_col
  MOV     r8, r12, LSR #3            ; r8 as des_row
  AND     r9, r12, #7                ; r8 as des_col

  SUB     r0, r6, r8
  BL      absolute
  MOV     r4, r0                    ; r4 as absolute(src_row - des_row), diff_row
  SUB     r0, r7, r9
  BL      absolute
  MOV     r5, r0                    ; r5 as absolute(src_col - des_col), diff_col
```

```
  ; start of switch
  MOV     r0, r11
  BL      get_cells_type
  CMP     r0, #'S'
  BEQ     legal_move_piece_s
  CMP     r0, #'P'
  BEQ     legal_move_piece_p
  CMP     r0, #'H'
  BEQ     legal_move_piece_h
  CMP     r0, #'B'
  BEQ     legal_move_piece_b
  CMP     r0, #'R'
  BEQ     legal_move_piece_r
  CMP     r0, #'Q'
  BEQ     legal_move_piece_q
  CMP     r0, #'K'
  BEQ     legal_move_piece_k
  B       legal_move_return_false

legal_move_piece_s:
  B       legal_move_return_false

legal_move_piece_p:
  CMP     r5, #1
  BGT     legal_move_return_false

legal_move_piece_p_non_forward_white:
  MOV     r0, r11
  BL      get_cells_side
  CMP     r0, #0
```

Game Logic – legal_move()

```
legal_move:
  STR     lr, [sp, #4]!            ; push lr
  BL      push_all_r0_r12
  MOV     r11, r0                  ; r11 as src_pos
  MOV     r12, r1                  ; r12 as des_pos
  MOV     r10, r2                  ; r10 as update

  MOV     r0, r11
  BL      is_own_piece
  CMP     r0, #0                   ; if(!is_own_piece(src_pos))
  BEQ     legal_move_return_false

  MOV     r0, r12
  BL      is_own_piece
  CMP     r0, #0                   ; if(is_own_piece(des_pos))
  BNE     legal_move_return_false

  CMP     r11, r12                 ; if(src_pos == des_pos)
  BEQ     legal_move_return_false

  MOV     r6, r11, LSR #3          ; r6 as src_row
  AND     r7, r11, #7              ; r7 as src_col
  MOV     r8, r12, LSR #3          ; r8 as des_row
  AND     r9, r12, #7              ; r8 as des_col

  SUB     r0, r6, r8
  BL      absolute
  MOV     r4, r0                  ; r4 as absolute(src_row - des_row), diff_row
  SUB     r0, r7, r9
  BL      absolute
  MOV     r5, r0                  ; r5 as absolute(src_col - des_col), diff_col
```

```
legal_move_piece_q:
  MOV     r0, r11
  MOV     r1, r12
  MOV     r2, #1
  MOV     r3, #1
  BL      is_path_clear
  CMP     r0, #0
  BEQ     legal_move_return_false
  BNE     legal_move_return_true
```

```
  ; start of switch
  MOV     r0, r11
  BL      get_cells_type
  CMP     r0, #'S'
  BEQ     legal_move_piece_s
  CMP     r0, #'P'
  BEQ     legal_move_piece_p
  CMP     r0, #'H'
  BEQ     legal_move_piece_h
  CMP     r0, #'B'
  BEQ     legal_move_piece_b
  CMP     r0, #'R'
  BEQ     legal_move_piece_r
  CMP     r0, #'Q'
  BEQ     legal_move_piece_q
  CMP     r0, #'K'
  BEQ     legal_move_piece_k
  B       legal_move_return_false

legal_move_piece_s:
  B       legal_move_return_false

legal_move_piece_p:
  CMP     r5, #1
  BGT     legal_move_return_false

legal_move_piece_p_non_forward_white:
  MOV     r0, r11
  BL      get_cells_side
  CMP     r0, #0
```

Thanks for watching!