EECS 373 Practice Midterm / Homework #3

Winter 2015

Due February 19th

Name: Key Uniquename: _____

Sign the honor code:

I have neither given nor received aid on this exam nor observed anyone else doing so.

_____

Problem #	Points
1	/10
2	/10
3	/15
4	/20
5	/15
6	/10
7	/10
8	/10
Total	**/100**

NOTES:

- **PUT YOU NAME/UNIQUENAME ON EVERY PAGE TO ENSURE CREDIT!**
- Can refer to the ARM Assembly Quick Ref. Guide and 1 page front/back cheat sheet
- Can use a basic/scientific calculator (but not a phone, PDA, or computer)
- Don't spend too much time on any one problem.
- You have 80 minutes for the exam.
- The exam is 9 pages long, including the cover sheet.
- Show your work and explain what you are doing. Partial credit w/o this is rare.

Name: Key uniqname: _____

1) **Fill-in-the-blank or circle the best answer. [10 pts, all or no points for each question]**

a) The ARM EABI specifies that registers r0-r3 are caller-save.

b) The ARM Thumb-2 instruction set has ***16-bit / 32-bit / 16- and 32-bit*** encodings.

c) To generate 8 ns pulses, a PWM controller with a 50 MHz clock needs a 40% duty cycle.

d) An ARM EABI-compliant procedure that calls another procedure should ***always / sometimes / never*** save and restore the `lr` register.

* e) ***UART / SPI / I2C*** supports flow control.

f) ***UART / SPI / I2C*** has built-in support for error checking by parity.

g) SPI bus transfers are ***asynchronous / synchronous*** and use ***dedicated chip select lines / addresses embedded in frames***.

h) In the Verilog hardware description language, an `@(` posedge clock `)` block would be used when implementing a flip-flop.

i) If multiple devices share a single interrupt line and generate interrupts at the same rate, then processor workload ***remains constant / grows linearly / grows quadratically*** with the number of devices.

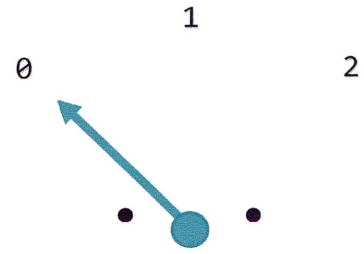
j) By default, uninitialized global variables go in the ***.text / .data / .bss*** section.

* This was a bad question (sorry). The DTS/CTS signals in UART are also a form of flow control (although less powerful than I2C's). We would accept either (or both) UART and I2C as answers.

Name: Key uniqname: _____

2) **Hardware Design. [pts]** 5 pts

Imagine an arm with 3 positions: 0, 1, and 2. A motor controls this arm. The motor can be driven forward by asserting MOTOR_FWD and in reverse by asserting MOTOR_REV. It is an error to assert both signals.



The arm position is measured by an encoder that reports current arm position as an 8-bit value. Mechanical stops prevent the arm from going past 0 to the left or 255 to the right. The positions are at encoder values 20, 127, and 235. You may assume that the arm has no inertia, that is, if neither MOTOR_FWD nor MOTOR_REV are asserted, the ARM_POSITION will not change.

Write a hardware module to control this arm. Your module should move the arm to the TARGET_POSITION ***only*** when the MOVE_REQ signal is asserted. POSITION will not change while MOVE_REQ is high. Your module should assert MOVE_DONE when the arm is in the desired position. It should then wait until MOVE_REQ is de-asserted and de-assert MOVE_DONE in response. MOVE_REQ will not re-assert until MOVE_DONE is de-asserted.

```
module arm_control (
        input CLOCK,

        // control interface
        input [1:0] TARGET_POSITION,
        input MOVE_REQ,
        output MOVE_DONE,

        // arm interface
        input [7:0] ARM_POSITION
        output MOTOR_FWD,
        output MOTOR_REV
)

`define POS_0 8'd20
`define POS_1 8'd127
`define POS_2 8'd235
```

always @(posedge CLOCK) begin
MOTOR_FWD <= next_FWD;
MOTOR_REV <= next_REV;
MOVE_DONE <= next_DONE;
end
(<= vs. =)

Must synchronize DONE signal to CLOCK (motor control could be combinational)

endmodule.

RUBRIC (5 pts)

-1 per error until 0.

if always need reg

```
reg MOVE_DONE, MOTOR_FWD, MOTOR_REV;
reg next_DONE, next_FWD, next_REV;

always @* begin
    next_FWD = 0;
    next_REV = 0;
    next_DONE = MOVE_DONE;

    if (MOVE_REQ && !MOVE_DONE) begin
        if (TARGET_POSITION == 2'b00) begin
            if (ARM_POSITION > `POS_0) next_REV = 1;
            else if (ARM_POSITION < `POS_0) next_FWD = 1;
            else next_DONE = 1;
        end
        --- same ---
    end
    if (MOVE_DONE && !MOVE_REQ) next_DONE = 0;
end

endmodule
```

must define in every path

only move if REQ

no FWD and REV = 1 case

assert DONE

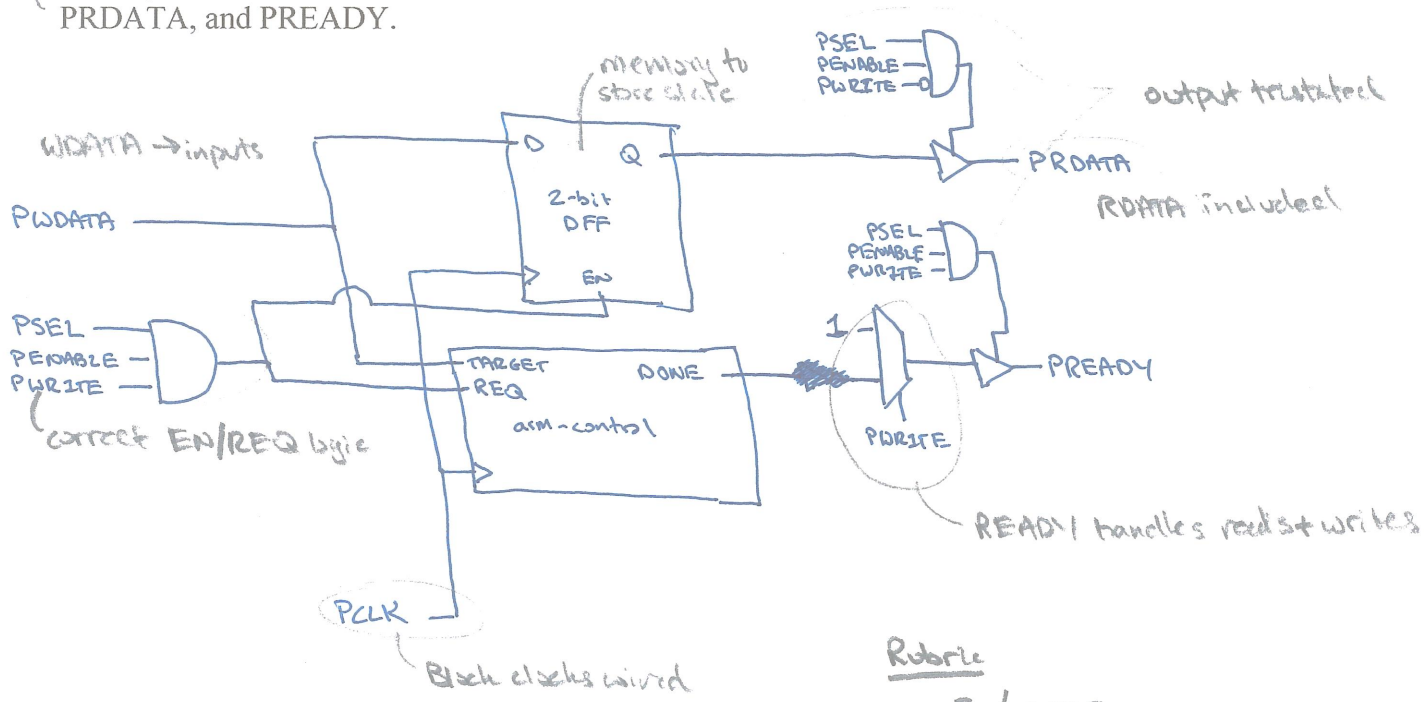
remember to deassert DONE

Name: Key uniqname: _____

3) **Memory-Mapped I/O.** ~~[pts]~~ 20 pts

10 pts

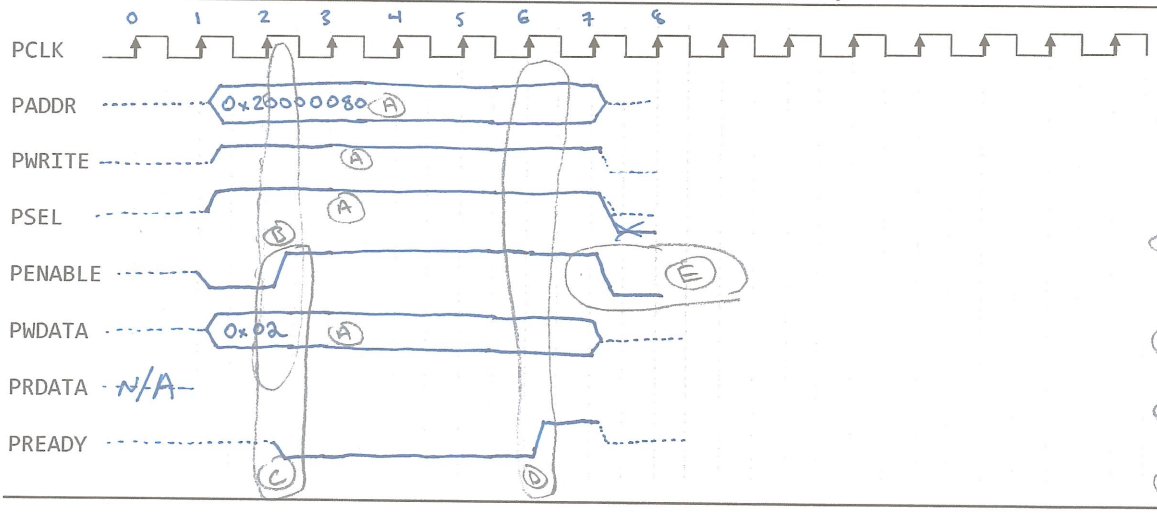
Using only basic circuit elements (e.g. AND and OR gates), sketch the glue logic required to interface the arm control module from Question 2 to the APB. Recall, the PSEL line is the peripheral select (i.e. it goes high when the processor is addressing the arm). The POSITION will be attached to data bits [1:0]. You should be able to change the arm position by writing to memory and read the current value by reading. A memory write should not return until the move is complete. The APB signals are: PCLK, PADDR, PWRITE, PSEL, PENABLE, PWDATA, PRDATA, and PREADY.



Rubric
-2 / error

5 pts

Sketch a timing diagram of an APB transaction that moves the arm from position 1 to position 2. Assume the arm peripheral is at address `0x20000080`. You only need to fill in relevant signals.



Rubric (1 pt/ea)

(A) PWRITE 1? DATA correct? ADDR correct? SEL 1?
(B) SEL, ADDR, DATA, WRITE set before PENABLE
(C) PREADY low in response to ENABLE
(D) PREADY assert before other changes
(E) End transaction by pulling ENABLE low

Name: Key uniqname: _____

4) ARM Assembly Language. 10 pts

Encoding T1 ARMv7-M
MOVT<c> <Rd>,#<imm16>

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1	1	1	1	0	i	1	0	1	1	0	0	imm4				0	imm3			Rd				imm8							

```
d = UInt(Rd);  imm16 = imm4:i:imm3:imm8;
if d IN {13,15} then UNPREDICTABLE;
```

Encoding T1 All versions of the Thumb instruction set.
MOVS <Rd>,#<imm8> Outside IT block.
MOV<c> <Rd>,#<imm8> Inside IT block.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Encoding T1 All versions of the Thumb instruction set.
STR<c> <Rt>,[<Rn>,<Rm>]

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	1	0	0	0	Rm			Rn			Rt		

```
t = UInt(Rt);  n = UInt(Rn);  m = UInt(Rm);
index = TRUE;  add = TRUE;  wback = FALSE;
(shift_t, shift_n) = (SRType_LSL, 0);
```

0101 00 1 000

a) What does the instruction 0x5088 do when executed?

2 pts

~~Store the value at MEM[R1 + R2]~~
Write the value of register R0 to the address R1+R2
1pt 1pt

b) Fill in machine code in hex around 0x5088 such that the arm peripheral from the previous question moves to position 1 when the instructions are executed.

_____
0x2200 MOVS, R2, #0
0x2180 MOVS, R1, #0x80
0xF2C20100 MOVT R1, 0x20000000
0x2001 MOVS R0, #1 0010 0 ddd 0000 0001
0x5088
_____
_____
_____
_____
_____

Note
- order (mostly) doesn't matter [except MOVT]
- Can flip r1 and r2

Rubric
2 pts Set R0 to 1
4 pts Set r1+r2 to 0x20000080
2 pts MOVT after MOVS

Name: Key uniqname: _____

5) **Assembly, C, and the ABI.** ~~[10]~~ 15 pts

Recall the arm peripheral is at address 0x20000010 and encodes the arm position in the bottom two data bits.

a.) Write a C function, `bool check_move(int change)`, that takes a move and returns 1 if it is a possible move and 0 if the move cannot be made. The change argument describes the change of arm position, e.g. change "2" would move an arm from position 0 to position 0+2 = 2 and change "-1" would return an error if the arm were in position 0. This function should ***not*** move the arm.

5 pts

```
bool check_move (int change) {
  int32_t current = *((int32_t *) 0x20000050);
  if ((current+change) > 2)
    return 0;
  if ((current+change) < 0)
    return 0;
  return 1;
}
```

-3 pts for correct read from MMIO address

2 pts for rest

b.) Write an assembly function `int move_arm(unsigned steps, bool reverse)` that moves the arm peripheral the requested number of steps forward or backward. Your function ***must*** call the `check_move()` function from (a) to verify if the requested move is valid before attempting to move the arm. `move_arm()` should return the current arm position, regardless of whether the arm moved or not.

10 pts

The assembly code should assemble without errors. Write clearly and comment the code.

```
label
move_arm: push {r4,r5,lr}        - callee-save
                                   -5 pts if no comments
          cmp  r1, #1            - use args from right regs.
synonym for IT   EQ              // if reverse, -steps
RSBEQ r0,#0 → NEGEQ r0           - handle reverse
          movs r5, r0            // save steps
          BL   check_move        // call check
                                   -5 pts if no attempt to call check_move
                                   (-1 for other errors)
load address
          movs r4, 0x80
          movt r4, 0x20000000    // load periph. address
          LDR  R1, [R4,#0]       // load arm pos.
          cmp  R0, 0
          ITT  EQ
          ADDEQ R1, R5           // conditionally store new arm position
          STREQ R1, [R4,#0]
          movs R0, R1            // set up return value + return.
          pop  {r4,r5,pc}        - callee-restore
                                   pop pc or bx lr to return
```

Rubric

-1 / error

Write absolute position, not steps

Write new arm pos.

Arm only moves if check_fn okays

Arm position is returned

return value is in R0.

This is 15 instructions. +1 EC for 13 insts, +3 EC for less than 13 insts. Must be correct for any EC.

Name: Key uniqname: _____

10pts 6) Write a C function `void enable_interrupts(int x)` that enables interrupt `x`. You need not check to validate that `x` is a legal interrupt number. The table below might be useful.

Table 6-1 NVIC registers

Address	Name	Type	Reset	Description
0xE000E004	ICTR	RO	-	*Interrupt Controller Type Register, ICTR*
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	0x00000000	Interrupt Set-Enable Registers
0xE000E180 - 0E000xE19C	NVIC_ICER0 - NVIC_ICER7	RW	0x00000000	Interrupt Clear-Enable Registers
0xE000E200 - 0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	RW	0x00000000	Interrupt Set-Pending Registers
0xE000E280 - 0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	RW	0x00000000	Interrupt Clear-Pending Registers
0xE000E300 - 0xE000E31C	NVIC_IABR0 - NVIC_IABR7	RO	0x00000000	Interrupt Active Bit Register
0xE000E400 - 0xE000E4EC	NVIC_IPR0 - NVIC_IPR59	RW	0x00000000	Interrupt Priority Register

```
void enable_interrupts(int x) {
  uint32_t *base = (uint32_t *) 0xE000E100;   <- 2pts for mmio syntax
  unsigned idx = x >> 5;
  unsigned bit = x & 0x1f;
  uint32_t mask = 1 << bit;
  *(base+idx) = mask;
}
```

4pts for correct mask
4pts for mask concept
(read old val, modify only part, write back)

Name: Key uniqname: _____

15 pts 7) **Startup and Interrupts.**

In addition to the motor, the system has two buttons. Button0 is wired to Interrupt 7 and Button1 is wired to Interrupt 8. Whenever Button0 is pressed, the arm should move left (decrement position). If Button1 is pressed, the arm should move right (increment position).

Using the functions from Questions 5 and 6, write any remaining code required such that the arm goes to position 1 when the system powers on and the buttons behave as intended.

```
// vector table
    .word   STACK_TOP      any name or reasonable value
1   .word   reset_fn
2   .word   reset_fn
3   .word   reset_fn       (don't care)
4   .word   reset_fn
5   .word   reset_fn
6   .word   reset_fn
7   .word   decr_fn
8   .word   incr_fn
```

5 pts for vector table
- 1 pt for stack top
- 2 pt reset handler
- 2 pt int7 / int8 handlers

-1 pt per minor error if appropriate

```
void reset_fn (void) {
    enable_interrupts (7);
    enable_interrupts (8);
    while (1) {
        __asm("wfi;");
    }
}
```

5 pts for main function
- 2 pts for enabling interrupts
- 2 pts for while loop (or other never-return idea)
- 1 pt for "wfi;" (or other non-busy wait idea)

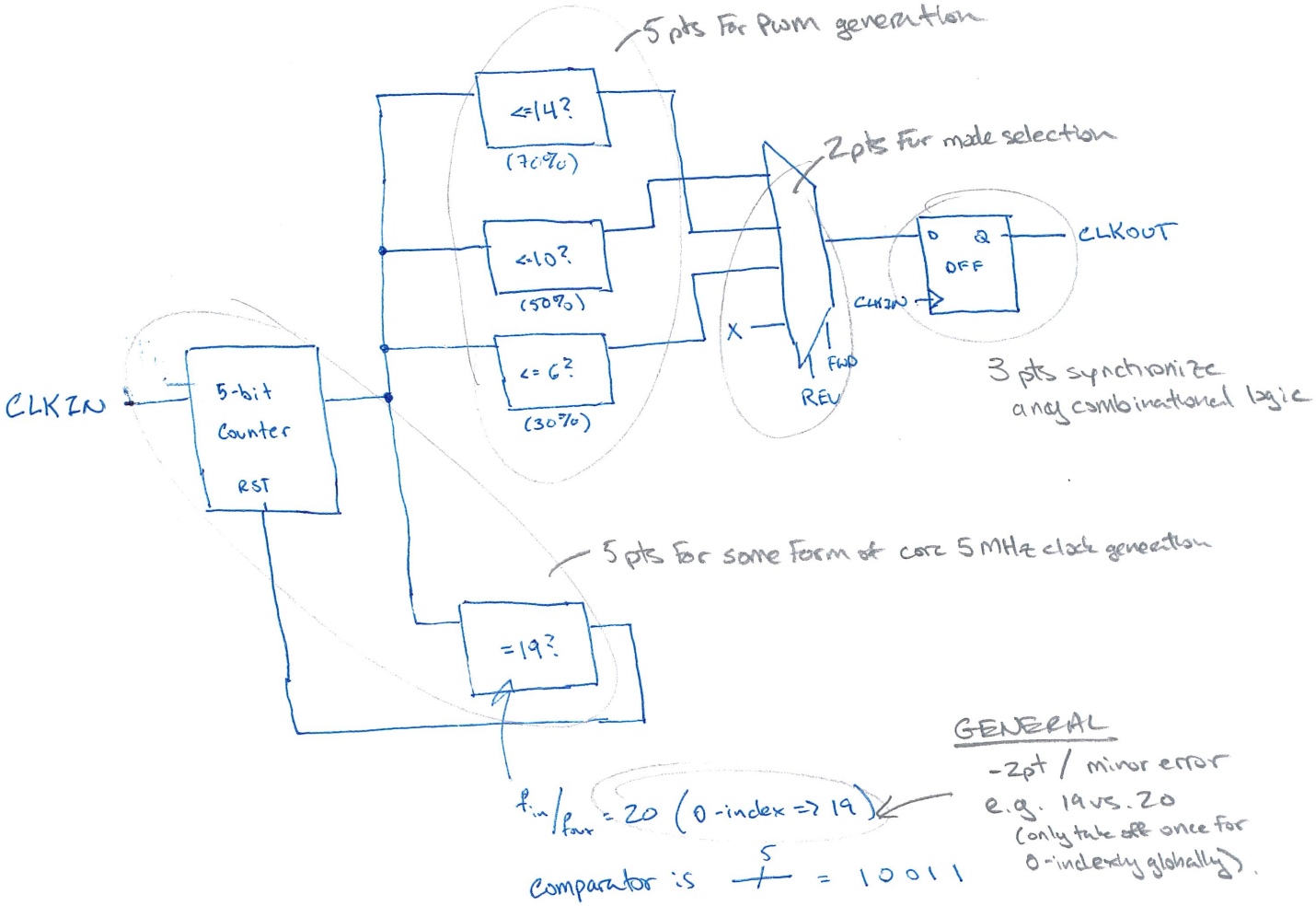
```
void incr_fn () {
    move_arm (1, 0);
}

void decr_fn() {
    move_arm (1, 1);
}
```

5 pts for interrupts handlers
- 2 pts / handler
- 1 pt for correct move_arm calls

Name: Key uniqname: _____

15 pts 8) **Logic Design [15 pts].**

The motor controller actually requires a PWM signal to operate. Given a 50% duty cycle, the motor will not move. At 70% it will drive forward, and at 30% it will drive in reverse. Design a digital circuit with inputs MOTOR_FWD and MOTOR_REV that takes the system clock of 100 MHz (CLKIN) and converts it to an output clock of 5 MHz with a 30, 50, or 70% duty cycle (MOTOR_CLK). You may use synchronously resettable D flip-flops and n-bit binary counters (make sure to specify the value of n). You may express combinational logic as Boolean expressions or using standard logic gates. You may assume that MOTOR_FWD and MOTOR_REV will never assert at the same time. Label things and write neatly.



```
# This is my 13-instruction answer to Q5b. I don't have a shorter solution.
#
# The tricks I used:
#  - Use a PC-relative load. The code will be the same size, but execute one
#    fewer instruction than a MOVW/MOVT pair.
#  - Optimizing for _instruction_ count, not _cycle_ count. That means that
#    pushing/popping multiple registers is free
#  - A few key properties:
#    -- Both success and error path must return current arm position
#    -- Want to end up with position in r0, but r0 also holds check_fn return
#    -- LDR instructions don't change the condition flags
#    -- A write to the arm position MMIO register will succeed
#    So, the code calls check_fn() and sets the condition flags based on the
#    result. It then gets the current arm position into r0. Then it actually
#    uses the condition result to optionally update the arm postion in r0 and
#    write that to the arm. Notice that the final arm position ends up in r0
#    through both the success and error path.

move_arm:
        push    {r4, r5, lr}

        cmp     r1, 1
        it      eq
        negeq   r0
        movs    r5, r0
        bl      check_fn
        cmp     r0, 1

        ldr     r4, addr
        ldr     r0, [r4, 0]

        itt     eq
        addseq  r0, r5
        streq   r0, [r4, 0]

        pop     {r4, r5, pc}

addr:   .word   0x20000080
¿
```