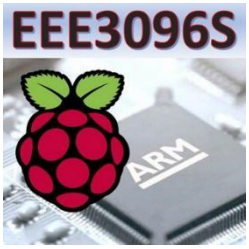


Embedded Systems II
EEE3096/5S

FINAL EXAM *14 October 2022 3 hours*

Examination Prepared by: Simon Winberg *Last Modified: 7-Nov-2022*

REGULATIONS

NB!

Dear Students,
Please note this is a **closed-book exam**. Provide your answers to the questions in the provided UCT examination booklet. You are recommended to detach the cheat sheets that are on the last pages of the exam paper. The exam is printed double sided. You are recommended to first scan through the questions quickly before starting, so that you can plan your strategy for answering the questions. If you are caught cheating, you will be referred to University Court for expulsion procedures. Make sure to **put** your **student name and student number** on your answer booklets. Please indicated at least your student number on the answer booklets with the **course code EEE3096S or EEE3095S** and **a title ES2 Final Exam**.

DO NOT TURN OVER UNTIL YOU ARE TOLD TO

Exam Structure

Marked out of 100 marks. 180 minutes (30min of which are in consideration for reading time needed).

RULES

- **Answer all questions; no questions are optional and will lose marks if no attempt is made to answer**
- Make sure that you cross out material you do not want marked. Your first attempt at any question will be marked if two answers are found.
- Use one of the answer book to plan the facts for your written replies to questions, so that you produce carefully constructed responses (then cross out rough work to avoid the possibility of the rough attempt being marked).
- Answer all questions, and note that the time for each question relates to the marks allocated (which implies there are a few minutes to allow you to think and shuffle papers etc).

Structure of exam:

C: Design Questions / problem-based scenarios to test your mettle

B: Short Answers

A: Multiple Choice

SECTION A:

MULTIPLE CHOICE AND SOME EASYPEASY QUESTIONS [5 x 8 = 40 marks]

Circle the letter a - d to select your choice of answer
(select only one answer option unless explicitly stated otherwise)

Question A.1. [5 marks]

The main differences between RISC and CISC processor instruction architectures were discussed in lectures. Which of the following points are correct in regards to these? *Note: You can select more than one option, as there may be more than one option that is correct, but a wrong selection may mean 0 for this question.*

Which of the following points are correct in regards to comparing RISC and CISC?

a) RISC has less instructions and therefore often needs more software code

b) RISC processors always outperforms the speed of CISC processors

c) CISC has more instructions, and more complex ones, thus often needs less software code

d) CISC has better load time of instructions as they are often shorter than RISC instructions

Question A.2. [5 marks]

Hopefully you can remember some digital logic, since it is extensively used in computer architecture… so you should manage this… In the circuit below, the inputs are 4-bit signals A and B and a 1-bit Sel signal[1].

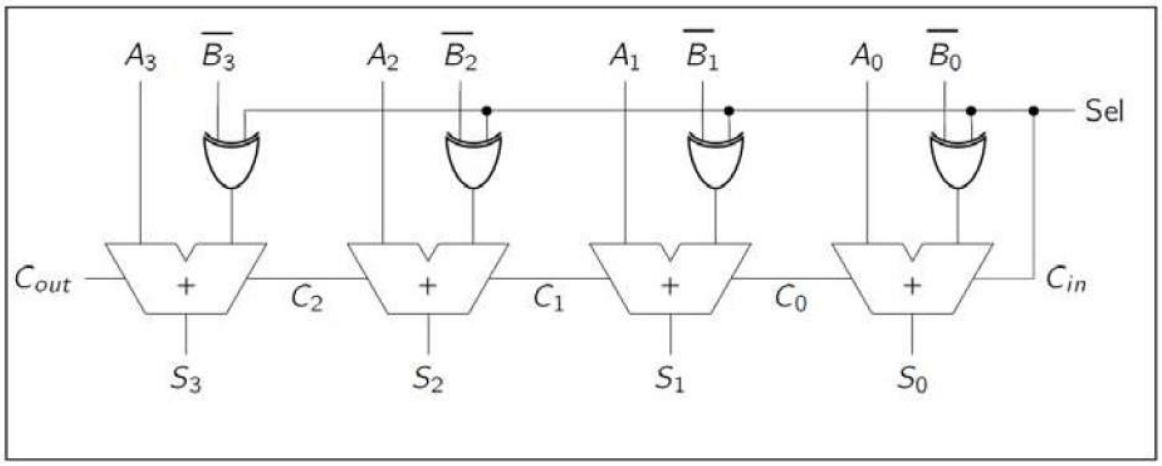


Figure 1: Circuit discussed in Question A.2

What operation happens when Sel = 1? Select *only one* option below indicating the result S gets set to:

a) S = A + B + 1
b) S = A – B
c) S = – (A + B)
d) S = – A + B

[1] you might remember the circuit, much the same, as selecting register to assigned a result to in the writeback stage for an ISA.

Question A.3. [5 marks]

There are various classifications for real-time systems. Which one of the following options best defines the characteristics of a periodic real-time system? (select only one option)

a) A system for which any one of its operations must complete within a specified time bound.

b) A system that meets (at least one) specific hard real-time deadline that repeats at a specific interval.

c) A system that provides real-time performance by means of polling instead of interrupts.

d) A system that has a collection of tasks that executes repeatedly at a specific period.

Question A.4. [5 marks]

Considering that the assembly function below is compatible with the C calling convention and compiler optimizations in use, what would be returned by myfunc(42) if called in a C function?

```
@ myfunc(unsigned int x):
myfunc:
        mov     r3, r0
        tst     r0, #1  @ test if first bit high
        bne     meexit
        movs    r0, #0
meloop:
        adds    r0, r0, #1
        lsrs    r3, r3, #1  @ shift
        tst     r3, #1
        beq     meloop
        bx      lr
meexit:
        movs    r0, #0
        bx      lr
```

What is the return of calling myfunc(42)? Select only *one option* below:

a) 0
b) 1
c) 2
d) 3

Question A.5. [5 marks]

The speed of ADC can vary substantiative according to their design, and their cost can vary quite a bit depending on their hardware complexity …

What is hardware the complexity of the standard Successive Approximation ADC design?
(Select only one option).

a) O(1)
b) $O(\log_2(n))$
c) O(n)
d) $O(n \cdot \log_2(n))$

Question A.6. [5 marks]

Consider that we want to implement a 4-bit up-counter with reset in Verilog. Select which code options below looks most correct in providing such an implementation. Select *only one option* for your answer.

a)

```
module up_counter (
   input clk, reset,
   output[4:0] counter  );
  reg [4:0] counter_up;

  // up counter
  always @(posedge clk or posedge reset)
    begin
     if (reset) counter_up <= 4'd0;
     else
     counter_up <= counter_up - 4'd1;
    end
  assign counter = counter_up;
endmodule
```

b)

```
module up_counter (
   input clk, reset,
   output[3:0] counter  );
  reg [3:0] counter_up;

  // up counter
  always @(posedge clk or posedge reset)
    begin
     if (reset) counter_up <= 4'd0;
     else
     counter_up <= counter_up + 4'd1;
    end
  assign counter = counter_up;
endmodule
```

c)

```
module up_counter (
   input clk, reset,
   output[4:0] counter  );
  reg [4:0] counter_up;

  // up counter
  always @(posedge clk)
   begin
    always @(posedge reset)
      counter_up <= 4'd0;
    if (reset==0)
      counter_up <= counter_up + 4'd1;
   end
  assign counter = counter_up;
endmodule
```

d)

```
module up_counter (
   input clk, reset,
   output[3:0] counter  );
  wire [3:0] counter_up;

  // up counter
  always @(clk or reset)
   begin
    if (reset)
      counter_up <= 4'd0;
      else
      counter_up <= counter_up + 4'd1;
   end
  assign counter = counter_up;
endmodule
```

Question A.7. [5 marks]

The ARM CPU can be configured to be little or big endian; it could make a mess of transferring results between processors through shared memory if you're expecting the one endianness instead of the other endianness. When in **big endian** mode, what does this mean in terms of how bytes within memory words are ordered, as in where the most significant and least significant bits of a data word are located?

a) Least Significant Byte is located at the lowest memory address
b) Least Significant Byte is located at the highest memory address
c) Most Significant Byte is located at the lowest memory address
d) Most Significant Byte is located at the highest memory address

Question A.8. [5 marks]

Consider that, in Verilog code, in our top-level module, we have two input ports, called A and B, that are each 8-bit buses, and we have another 5-bit register called N. We want to combine A and B into a single 16-bit bus that we will hook up to the first port, which is 16-bits, of another module, called *hibitcount*. The *hibitcount* has a second port, a 5-bit output port, that will be set to the count of how many bits in the first port are high. It doesn't matter if A or B is the MSB as the module only counts the number of bits that are high on its first port. It is important that only zeros (i.e. bits tied to 0) are added if needing to pad additional bits to the input. Which option below would achieve this? Assume N is already declared as a 5-bit register in the top-level module. Select only *one* option for your answer:

a) hibitcount([A,B],N);
b) hibitcount({A,B},N);
c) hibitcount(A:B,N);
d) N = hibitcount(A,B);

SECTION B:

SHORT ANSWERS – NOT QUITE SO EASY QUESTIONS [30 marks]

Question B1 [10 marks]

B1.1. If you had to choose between using SPI or using I2C as a communication protocol, what would influence your choice in deciding which protocol to use? Name at least three reasons (besides cost) for choosing the one over the other.

[5 marks]

B1.2. Consider that you're working at a development company, and you're in deep discussion with a trainee engineer, explaining the communication needs for a prototype, which you are hoping to delegate to the trainee. You're rapidly using terms like 'USB', 'RS232' and 'UART' in your explanations. But suddenly there's a (metaphorical) spanner in the works: the trainee interrupts your narrative asking: "But sir, what is RS232 and UART? Are those like robots, where should I get those from?" Try to provide a gentle but accurate response to clarify the concepts of RS232 and UART in regards to embedded system design.

[5 marks]

Question B2 [12 marks]

The assembly code for the *checkbalance* function, and the main function that calls this assembly routine, is given below, this code relates to both sub-questions B2.1 and B2.3. Start by having a look through the code to attempt to make sense of what is happening, consider that the parameters are passed in registers (r0 and r1) and that a stack is not used in passing parameters.

Assembly file: checkbalance.S

```
@ fast routine to check how balanced charging is of batteries 1 and 2
@ checkbalance(unsigned char* adc1, unsigned char* adc2):
checkbalance:
        ldrb    r0, [r0]        @ get value from adc1
        ldrb    r3, [r1]        @ get value from adc2
        cmp     r0, r3          @ compare values
        bgt     ret1
        cmp     r0, r3
        mov     r0, #-1
        it      ge              @ alerts GAS that ge flag expected not
                                @ to change in the previous instruction[2]
        movge   r0, #0
        bx      lr
ret1:
        movs    r0, #1
        bx      lr
```

(P.T.O for main.c program)

[2] This is generally the case, that MOV instructions on ARM don't change flags since the data doesn't go via the ALU.

C file: main.c

```
/* Program that does various operations, in do_stuff() and then
   checks how evenly the system's batteries are getting charged. */

#include <stdio.h>

unsigned char* ADC1 = 0xFFFFFF10; /* address of memory-mapped ADC1 */
unsigned char* ADC2 = 0xFFFFFF20; /* address of memory-mapped ADC2 */

/* start ms timer, implemented in hwtimers.c */
extern void start_timer();

/* returns number ms since start_timer called, see hwtimers.c */
extern unsigned read_timer();

/* does other processing that the program needs to do */
void do_stuff();

/* absolute function ... given to reassure yourself what it does */
int abs (int x) {
     if (x<0) return -x;
     return x;
}

/* main function that implements a continuous while loop */
int main ()
{
  /* set pointers adc1 and adc2 to point respectively to the
     ADC measuring battery1 and the ADC measuring battery2. */
  unsigned char* adc1 = ADC1;
  unsigned char* adc2 = ADC2;
  int chargebal = 0;
  printf("monitor charging balance\n");

  /* main loop for the program */
  while (1) {
     start_timer(); /* start the ms timer that counts elapsed time
                       in ms from now */
     do_stuff();    /* do various polling and processing */
     /* check the charging of the batteries before waiting for
        the next poll period */
     chargebal = chargebal + checkbalance(adc1,adc2);
     /* report charging inbalance if after a couple seconds it is
        found that the one battery charges faster than the other
        over a few seconds */
     if (abs(chargebal)>400) {
          printf("charge speed discrepancy of %d\n",chargebal);
          chargebal = 0;
     }
     /* just wait in a tight look until 100 has elapsed since
        start_timer() was called */
     while (read_timer()<100);
  }

  return 0; /* shouldn't get here */
}
```

B2.1. Consider that the c code documenting what checkbalance.S is doing has been lost. Review that assembly routine and explain what is happening. You can provide pseudo code – or ideally equivalent C – to explain the routine.

[6 marks]

B2.2. If the program keeps reporting a charge speed discrepancy that is a negative number, then what could this possibly mean in terms of how the batteries are charging?

[3 marks]

B2.3. If you needed to time, in milliseconds, how long `do_stuff()` took to complete, how could you go around determining that? Explain how you might change or add to the code to provide that facility.

[3 marks]

Question B3 [4 x 2marks = 8 marks]

At last, the second most desired moment you've been awaiting: the TRUE / FALSE questions!
(The most desired moment is probably finishing the exam).

Answer the following statements as **true** to indicate that the statement is correct, or **false** to indicate the statement is not correct. Answer only yes or no for each statement, selecting both will get 0.

B3.1. The ARM processors can only run programs compiled in C. True / False ?

B3.2. It was mentioned in the CPU architecture slides that the computation a processor is to carry our can be represented as spatial computation, which is a paradigm of representation focusing on how operations are sequenced in time. True / False ?

B3.3. A datapath can be considered the collection of the registers, processing units, and interconnections used to process and transfer data in a computer system. True / False ?

B3.4. The commonly used 'Brisbane definition' of an embedded system centers around the concept that an embedded system is task-specific computer and usually built into a larger system. True / False ?

SECTION C:

TESTING YOUR METTLE QUESTIONS [30 marks]

The Following Scenario Relates to Questions C.1 and C.2 that follow

Take a bit of time to read this scenario and design concept; the time for the exam is planned on you taking about 15 minutes reading and making sense of the scenario, and then answer the questions that follow.

Scenario:
"Happy Fit Step Pedmeter" – The Remarkable Sound Pedometer[3]

A pedometer is a wearable device that counts the number of steps the user takes. It is generally quite a simple operation, just using an accelerometer that is configured to sense a step opposed to someone just typing at a keyboard. The pedometer system concerned has two lousy push buttons and a scratchy piezoelectric speaker as an interface (yes, indeed it was that online one that looked so cool on the web and promised 'Happiness Greetings!!' and 'Good Smiling Sound if walk longtime').

It works as follows:

- Initially (on boot, i,e, first time turned on) today_steps and hour_steps are both reset to 0, and it (as you suspect) pays back a greetings tune. The time is also set to 00:00, i.e. minutes 0 and hours 0.
- The time register is updated every minute (via the RTC), i.e. minutes increments every minute but when at 59 changes to 0 and increments the hours. When the hours is at 23 and is incremented, it wraps to 0.
- If button 1 is pressed then minutes increments (wrapping to 0 if at 59).
- If button 2 is pressed then hours is incremented (wrapping to 0 if at 23)
- If the time equals to 07h00 then the pedometer plays "Good morning!" on its speaker[4].
- Whenever a step (posedge step) is sensed then today_steps increments and hour_steps increments
- If minutes changes to 00 then hour_steps is set to 0
- If minutes equals 50 and hour_steps < 200 then plays "Step Time!" loudly (to disturb meetings)
- If time equals 17h00 and (today_steps > 8000) then plays "Healthy Day!"
- If time equals 00h00 then hour_steps and today_steps are both set to 0

[3] This being the verbatim wording from the hypothetical product box.
[4] To get a good sense of the user experience with this product, imagine a 2Kb 1-bit PWM recording of 'Good Morning' being played back on a piezoelectric speaker at 7am on a weekend morning.

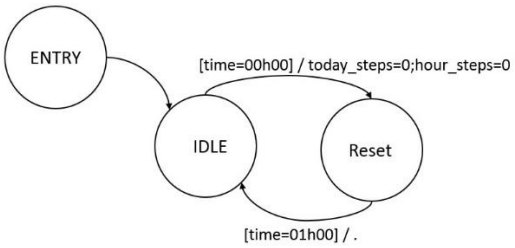
Question C.1. [15 marks]

This question concerns a combination of finite state machine modelling and Verilog. First read the scenario on the previous page before attempting this question.

Complete the following

a) A starting point has been given below for your state chart. What is wrong with this design or is missing in regard to the operation description above? Briefly explain two things at least.

[4 marks]



b) Complete the state chart according to the description

[11 marks]

Inputs and output functions available
time : this return a quantity XXhYY where XX indicates the hours elapsed since 00h00 and YY indicates the minutes elapsed since the hour changed (assume Time links to a RTC device to return the actual time).
minutes : this returns just the number of minutes elapsed since the hour changed.
hours : this returns just the number of hours elapsed since 00h00.
hour_steps : number of steps done this hour
today_steps : number of steps done today since 00h00
step: a pulse (change from low to high for, a high lasting for 1us) caused when a step is detected
Play(string) : You can assume this function will play a sound (e.g. Play("file1") will play file1.wav).

Question C.2. [15 marks]

This involves a short bit of Verilog programming, to program the `detect_step` module introduced on the next page. For this you are to implement a state machine in Verilog for detecting a step sing the accelerometer. The step output is usually kept low (0), and should be set to 0 on reset. When a step is detected the step output line should be help high (1) for 1us and then set low. See below the overview of how a step is detected using the accelerometer concerned.

You have the following input and output ports (you can choose to add more):
input reset : this is raised high and reset hour_steps to zero and today_steps to zero.
input clk : this is the 1us input clock to the module
input accd : this is a 3 bit input that indicates the direction of maximum acceleration angle
000 = no acc 001= x dir, 010= y dir, 100= z dir , 011=y&x dir, 101=z&x dir, 110=z&y dir
input accm : this is a 5 bit input that indicates the 4-bit magnitude of the maximum acceleration angle and the MSB bit (i.e. 10000_2) indicates the sign, 0 or positive and 1 for negative)
output step: indicate step occurred (hold high for 1us and return to 0 to indicate step step was detected

See next page for method of how to detect a step.

Method of detecting a step

The pedometer has an accelerometer that has two digital outputs: *accd* and *accm*.
The *accd* is a 3-bit bus. It is used to indicate the direction of maximum motion; the bits (from left-most least significant to right-most bit) indicates if the motion is partly in the x direction (001_2), the y direction (010_2) or z direction (100_2). A 0 in a axis direction indicates there is no forward or back motion in that direction. For *accm*, this indicates the magnitude of the motion, which is a 5-bit bus, the left-most bit indicates the sign for the direction, if forward or backwards, and the next four bits indicates the magnitude. So a value of accd=010_2 and accm=10001_2 would be a gentle upwards motion as the dominant, force; whereas accd=010_2 and accm=01111_2 would be a strong downwards force being dominant.

So, the way to detect a step is to find a sudden, fairly noticeable, reverse in direction. For example if you have accd=010_2 (up/down) of a non-negligible force amplitude, e.g. accm=00010_2 or more, and a sudden reverse (as in a few 1000us later, otherwise it's just noise) in the dominant force angle, i.e. accd=010_2 and accm=10010 (remember MSB=sign bit), then a likely step has been made (it could be a lift stopping at a floor, or similar external factor causing the effect). So, in that situation of detecting a step, the detect_step module should put a brief pulse on the its step output line.

TODO: Complete the following:

a) You are given this starting point to the detect_step module and you need to implement functionality for producing a 1us step pulse output when a step is detected.

```
module detect_step (reset,clk,acca,accd,step);
   // add your code here:
   // define ports as inputs etc.
   input  reset; // indicates reset (if 1) is in process
   input  clk;   // 1us clock
   output step;  // generate a pulse on here
   // … Add any other inputs or registers that may be needed

   // implement some reset functionality
   // implement counting up of steps
   // implement the Step Time! Operation by raising playst
endmodule
```

[12 marks]

b) Briefly explain what a testbench is (1 mark). Discus how you would go about setting up a testbench for the *detect_step* module. (2 marks). You do not need to provide code for a testbench, you can focus on describing how you would use a testbench to test your module (you can provide snippets of Verilog or just pseudocode to aid your explanation).

[3 marks]

END OF EXAMINATION

Appendix A: Verilog Cheat sheet

Numbers and constants

Example: 4-bit constant 10 in binary, hex and in decimal: 4'b1010 == 4'ha -- 4'd10

(numbers are unsigned by default)

Concatenation of bits using {}

4'b1011 == {2'b10 , 2'b11}

Constants are declared using parameter:

parameter myparam = 51

Operators

Arithmetic: and (+), subtract (-), multiply (*), divide (/) and modulus (%) all provided.

Shift: left (<<), shift right (>>)

Relational ops: equal (==), not-equal (!=), less-than (<), less-than or equal (<=), greater-than (>), greater-than or equal (>=).

Bitwise ops: and (&), or (|), xor (^), not (~)

Logical operators: and (&&) or (||) not (!) note that these work as in C, e.g. (2 && 1) == 1

Bit reduction operators: [n] n=bit to extract

Conditional operator: ? to multiplex result

Example: (a==1)? funcif1 : funcif0

The above is equivalent to:

```
((a==1) && funcif1)
|| ((a!=1) && funcif0)
```

Registers and wires

Declaring a 4 bit wire with index starting at 0:

wire [3:0] w;

Declaring an 8 bit register:

reg [7:0] r;

Declaring a 32 element memory 8 bits wide:

reg [7:0] mem [0:31]

Bit extract example:

r[5:2] returns 4 bits between pos 2 to 5 inclusive

Assignment

Assignment to wires uses the assign primitive outside an always block, e.g.:

assign mywire = a & b

Registers are assigned to inside an always block which specifies where the clock comes from, e.g.:

```
always@(posedge myclock)
   cnt = cnt + 1;
```

Blocking vs. unblocking assignment <= vs. =

The **<=** assignment operator is non-blocking (i.e. if use in an always@(posedge) it will be performed on every positive edge. If you have many non-blocking assignments they will all be updated in parallel. The <= operator must be used inside an always block – you can't use it in an assign statement.

The blocking assignment operator **=** can be used in either an assign block or an always block. But it causes assignments to be performed in sequential order. This tends to result in slower circuits, so avoid using it (especially for synthesized circuits) unless you have to.

Case and if statements

Case and if statements are used inside an always block to conditionally update state. e.g.:

```
always @(posedge clock)
  if (add1 && add2) r <= r+3;
  else if (add2) r <= r+2;
  else if(add1) r <= r+1;
```

Note that we don't need to specify what happens when add1 and add2 are both false since the default behavior is that r will not be updated. Equivalent function using a case statement:

```
always @(posedge clock)
  case({add2,add1})
  2'b11  : r <= r+3;
  2'b10  : r <= r+2;
  2'b01  : r <= r+1;
  default: r <= r;
endcase
```

Module declarations

Modules pass inputs, outputs as wires by default.

```
module ModName (
  output reg [3:0] result,  // register output
  input [1:0] bitsin,  input clk, inout bidirectnl  );
 … code …
endmodule
```

Verilog Simulation / ISIM commands

```
$display ("a string to display");
$monitor ("like printf. Vals: %d %b", decv,bitv);
#100   // wait 100ns or simulation moments
$finish  // end simulation
```

Appendix B: ARM Assembly Language Cheatsheet

Memory access instructions

```
LDR Rd, [Rn]            ; load 32-bit number at [Rn] to Rd
LDR Rd, [Rn,#off]       ; load 32-bit number at [Rn+off] to Rd
STR Rt, [Rn]            ; store 32-bit Rt to [Rn]
STR Rt, [Rn,#off]       ; store 32-bit Rt to [Rn+off]
PUSH {Rt}               ; push 32-bit Rt onto stack
POP  {Rd}               ; pop 32-bit number from stack into Rd

MOV{S} Rd, <op2>        ; set Rd equal to op2
MOV Rd, #im16           ; set Rd equal to im16, im16 is 0 to 65535
MVN{S} Rd, <op2>        ; set Rd equal to -op2
```

Branch instructions

```
B label                 ; branch to label Always
BEQ label               ; branch if Z == 1 Equal
BNE label               ; branch if Z == 0 Not equal
BCS label               ; branch if C == 1 Higher or same, unsigned ≥
BHS label               ; branch if C == 1 Higher or same, unsigned ≥
BCC label               ; branch if C == 0 Lower, unsigned <
BLO label               ; branch if C == 0 Lower, unsigned <
BMI label               ; branch if N == 1 Negative
BPL label               ; branch if N == 0 Positive or zero
BVS label               ; branch if V == 1 Overflow
BVC label               ; branch if V == 0 No overflow
BHI label               ; branch if C==1 and Z==0 Higher, unsigned >
BLS label               ; branch if C==0 or Z==1 Lower or same, unsigned ≤
BGE label               ; branch if N == V Greater than or equal, signed ≥
BLT label               ; branch if N != V Less than, signed <
BGT label               ; branch if Z==0 and N==V Greater than, signed >
BLE label               ; branch if Z==1 or N!=V Less than or equal, signed ≤
BX Rm                   ; branch indirect to location specified by Rm
BL label                ; branch to subroutine at label
BLX Rm                  ; branch to subroutine indirect specified by Rm
```

Logical instructions

```
AND{S} {Rd,} Rn, <op2>  ; Rd=Rn&op2 (op2 is 32 bits)
ORR{S} {Rd,} Rn, <op2>  ; Rd=Rn|op2 (op2 is 32 bits)
EOR{S} {Rd,} Rn, <op2>  ; Rd=Rn^op2 (op2 is 32 bits)
BIC{S} {Rd,} Rn, <op2>  ; Rd=Rn&(~op2) (op2 is 32 bits)
ORN{S} {Rd,} Rn, <op2>  ; Rd=Rn|(~op2) (op2 is 32 bits)
LSR{S} Rd, Rm, Rs       ; logical shift right Rd=Rm>>Rs (unsigned)
LSR{S} Rd, Rm, #n       ; logical shift right Rd=Rm>>n (unsigned)

ASR{S} Rd, Rm, Rs       ; arithmetic shift right Rd=Rm>>Rs (signed)

ASR{S} Rd, Rm, #n       ; arithmetic shift right Rd=Rm>>n (signed)
LSL{S} Rd, Rm, Rs       ; shift left Rd=Rm<<Rs (signed, unsigned)
LSL{S} Rd, Rm, #n       ; shift left Rd=Rm<<n (signed, unsigned)
```

Arithmetic instructions

```
ADD{S} {Rd,} Rn, <op2>   ; Rd = Rn + op2
ADD{S} {Rd,} Rn, #im12   ; Rd = Rn + im12, im12 is 0 to 4095
SUB{S} {Rd,} Rn, <op2>   ; Rd = Rn - op2
SUB{S} {Rd,} Rn, #im12   ; Rd = Rn - im12, im12 is 0 to 4095
RSB{S} {Rd,} Rn, <op2>   ; Rd = op2 - Rn
RSB{S} {Rd,} Rn, #im12   ; Rd = im12 – Rn
CMP Rn, <op2>            ; Rn – op2 sets the NZVC bits
MUL{S} {Rd,} Rn, Rm      ; Rd = Rn * Rm signed or unsigned
```

Notes

Ra Rd Rm Rn Rt represent 32-bit registers
value any 32-bit value: signed, unsigned, or address
{S} if S is present, instruction will set condition codes
#im12 any value from 0 to 4095
#im16 any value from 0 to 65535
{Rd,} if Rd is present Rd is destination, otherwise Rn
#n any value from 0 to 31
#off any value from -255 to 4095
label any address within the ROM of the microcontroller
op2 the value generated by <op2>