Two hours - online

EXAM PAPER MUST NOT BE REMOVED FROM THE EXAM ROOM

"ARM Instruction Set Summary" is attached

**UNIVERSITY OF MANCHESTER**
**DEPARTMENT OF COMPUTER SCIENCE**

Fundamentals of Computer Architecture

Date: Monday 13th January 2020

Time: 09:45 - 11:45

---

**This is an online examination. Please answer ALL Questions**
**The examination is worth a total of 80 marks**
**The exam contains MULTIPLE CHOICE QUESTIONS**



---

This is a CLOSED book examination

The use of electronic calculators is NOT permitted

**Section A**

# Section A contains restricted multiple choice questions (MCQs) and is NOT published.

## Section B

2. a) Explain briefly the following addressing modes.

   i) Direct addressing
   ii) Indirect addressing
   iii) Pre-index addressing
   iv) Post-index addressing (4 marks)

   b) Explain the differences between the following addressing modes and when to use one over the other.

   i) Direct addressing versus Indirect addressing
   ii) Base+Offset addressing versus Pre-Index addressing (4 Marks)

   c) Translate the following Python statements, which are just part of a much larger program, into an equivalent sequence of ARM instructions. You should assume that the integer variables x and y are in memory. Initially, upon program loading, the value of x is 0 and the value of y is 0. (10 points)

```
y= 10
while (x<y):
        if (x==0):
                x = x + 1
        else:
                x = x * 2
```

(12 Marks)

Total Marks for Q2: 20

[PTO]

3. a) Describe two techniques that could be used for transferring characters from a keyboard to the computer, giving an advantage and a disadvantage for each. (10 marks)

   b) Explain what is the stack and why it is important. (3 Marks)

   c) Explain two instructions that can manipulate the stack by providing two simple examples of their usage. (4 marks)

   d) Which register does the ARM architecture use to remember where to return after executing a method? Explain how this register is used to control the execution of a program by providing an example. (3 marks)

Total Marks for Q3: 20

4. a) Describe two different ways of translating a piece of python code, having a large number of nested if-else statements, into an equivalent ARM assembly code. Explain the advantages and disadvantages of each method. (10 marks)

   b) Instruction set architectures can be categorised based on the type of internal storage in a processor. Write assembly-like (not strict ARM assembly) code for the following expression for the "accumulator" and "register-memory" architectures where A, B and C are integer variables stored in memory. How many times the processor will access memory to perform this expression?

   B = A + C * C

(10 Marks)

Total Marks for Q4: 20

**END OF EXAMINATION**

# COMP15111 ARM instruction set summary

This is not a complete list and you may need to consult other documentation for more detail.

### Data Processing

| | | |
|---|---|---|
| ADD | Rd, Rn, Op | Rd= Rn + Op |
| SUB | Rd, Rn, Op | Rd= Rn – Op |
| RSB | Rd, Rn, Op | Rd= Op – Rn ("reverse subtract") |
| AND | Rd, Rn, Op | Rd= Rn AND Op |
| ORR | Rd, Rn, Op | Logical OR |
| EOR | Rd, Rn, Op | Exclusive OR |
| BIC | Rd, Rn, Op | Bit Clear: Rd= Rn & !Op |
| MOV | Rd, Op | Rd= Op |
| MVN | Rd, Op | Rd= !Op |
| CMP | Rn, Op | set status on Rn – Op |
| CMN | Rn, Op | set status on Rn + Op |
| TST | Rn, Op | set status on Rn AND Op |
| TEQ | Rn, Op | set status on Rn EOR Op |
| MUL | Rd, Rn, Rs | Rd= Rn * Rs |
| MLA | Rd, Rn, Rs, Rp | Rd= (Rn * Rs) + Rp |

An "Op" in the table above can be a literal (e.g. `#10`) or a register (e.g. `R1`) or a shifted register.

If a shift (by a literal e.g. `#3` or a register e.g. `R6`) is required it is specified as the fourth parameter e.g.
`ADD R1, R4, R5, LSL #3` ; R1 = R4 + R5*8
`ADD R1, R4, R5, LSL R6` ; R1 = R4 + R5*$2^{R6}$

| | |
|---|---|
| LSL Shift | logical shift left by $0 \leq Shift \leq 31$ places, putting 0s into least significant end |
| LSR Shift | logical shift right by $0 \leq Shift \leq 32$ places, putting 0s in most significant end |
| ASR Shift | arithmetic shift right by $0 \leq Shift \leq 32$ places,<br>copying the sign bit into the most significant end |
| ROR Shift | circular rotate right by $0 \leq Shift \leq 32$ places,<br>moving bits lost from one end into other end |
| RRX | one place right shift. Carry from status reg. shifts into most significant bit.<br>If status reg. set by instruction, carry bit = least significant bit.<br>This gives a 1-bit circular rotate through the carry bit |

### Loads and Stores

| | | |
|---|---|---|
| LDR | Rd, Address | loads a 32-bit word into Rd from memory location |
| LDRB | Rd, Address | loads 8-bits into Rd; top 24 bits are '0's |
| STR | Rd, Address | stores Rd at memory location |
| STRB | Rd, Address | stores bottom byte of Rd at memory location |
| LDMFD | Rd, register list | multiple reg load,load from addr Rd |
| STMFD | Rd!, register list | multiple reg store, Rd is updated after each store operation |

An "Address" in the table above can be a numerical address (e.g. `100`) or a named memory location (e.g. `fred`) or calculated from the contents of registers and literals e.g.:

| operand form | address | final value of R0 |
|---|---|---|
| [*R0*] | R0 | (unchanged) |
| [*R0*, *R1*] | R0 + R1 | (unchanged) |
| [*R0*, #1] | R0 + 1 | (unchanged) |
| [*R0*], *R1* | R0 | R0 + R1 |
| [*R0*], #1 | R0 | R0 + 1 |
| [*R0*, *R1*]! | R0 + R1 | R0 + R1 |
| [*R0*, #1]! | R0 + 1 | R0 + 1 |

In the examples above the value copied from `R1` can be modified by a shift (but `R1` itself is unchanged) e.g.
`LDR R2, [R0, R1, LSL #3]` ; address = R0 + 8*R1
The possible shifts are LSL, LSR, ASL and ROR as described for Data Processing instructions above.

*Please Turn Over*

# COMP15111 ARM instruction set summary continued

### Control Transfer

| | | |
|---|---|---|
| B | Label | branch to label: R15= label |
| BL | Method | branch and link: R14= R15, R15= method |
| SWI | Number | software interrupt |

### Condition Codes

Any instruction can be made conditional e.g. `ADD` can become `ADDEQ` etc.

| Code | Meaning | Flag condition |
|---|---|---|
| EQ | Equal | Z |
| NE | Not Equal | !Z |
| GE | Greater than or Equal (signed) | N = V |
| GT | Greater Than (signed) | (N = V) . !Z |
| LT | Less Than (signed) | N != V |
| LE | Less than or Equal (signed) | (N != V) + Z |
| HI | Higher (unsigned) | C . !Z |
| LS | Lower or Same (unsigned) | !C + Z |
| CS/HS | Carry Set/Higher or Same (unsigned) | C |
| CC/LO | Carry Clear/Lower (unsigned) | !C |
| MI | Minus (negative) | N |
| PL | Plus (positive) | !N |
| VS | Overflow Set | V |
| VC | Overflow Clear | !V |
| AL | Always | TRUE |

Also, any Data Processing instruction can be followed by an "S" e.g. `ADD` can become `ADDS` meaning that the result of the instruction is to be compared with zero.

### Assembler Supplied 'Pseudo Operations'

| | | |
|---|---|---|
| NOP | | no operation |
| MOV | Rd, #-nn | replaced by MVN |
| CMP | Rn, #-nn | replaced by CMN |
| ADR | Rd, label | Rd=label address. replaced by ADD Rd, R15, #nn |
| ADRL | Rd, label | Rd=label address. Used when #nn in ADR is out of range |
| LDR | Rd, =nnnnnnnn | Load constant |

### Directives

| | | |
|---|---|---|
| ORIGIN | &address | Set address of code following |
| ALIGN | | to next 4-byte boundary |
| DEFW | &12345678 | Define the value of the next word(s) |
| DEFB | 0, 2, "bytes" | Define the value of the next byte(s) |
| DEFS | &20 | Reserve the next 20 bytes |

DEFB, DEFW and DEFS allow data to be planted amongst the instructions in a program.