**Cortex M0 assembly summary**

Wouter van Ooijen

2018-7-18

A paper copy of this summary is provided with the V2CPSE1-16 exam.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Low  registers | |  |  |  |  | |
|  |  | R0 |  | Arguments,  Return value,  Scratch  (not preserved) | |
|  | R1 |
|  | R2 |
|  | R3 |
|  | R4 |  | Local variables (preserved) | |
|  | R5 |
|  | R6 |
|  | R7 |
|  |  | |  | R8 |
|  |  | |  | R9 |
|  |  | |  | R10 |
|  |  | |  | R11 |
|  |  | |  | R12 |  | reserved | |
|  |  | |  | R13 |  | Stack Pointer | |
|  |  | |  | R14 |  | Link Register | |
|  |  | |  | R15 |  | Program Counter | |
|  |  | |  |  |  |  | |
| **Segment / Region** | | | **Used …** | | | | | **Present in executable** | | **Allocated at** | **Allowed operations** |
| CODE | | | To store executable code | | | | | Yes | | Build | Execute |
| RODATA | | | To store read-only data | | | | | Yes | | Build | Read |
| DATA | | | To store data that has an initial value, but can be changed | | | | | Yes | | Build | Read, write |
| BSS | | | To store data that has no specific initial value | | | | | No | | Build | Read, write |
| STACK | | | For or the stack (local data and return addresses) | | | | | No | | Run | Read, write |
| HEAP | | | For the heap (dynamically allocated data) | | | | | No | | Run | Read, Write |

|  |  |
| --- | --- |
| **Directive** | **Effect** |
| .align N | Skip addresses, until the address is divisible by N. |
| .ascii ”text”, ”text2”, … | Produce the bytes that contain the ASCII characters in each string. |
| .asciz ”text”, ”text2”, … | As .ascii, but add a ’\0’ byte after each string. |
| .bss | Following items are produced in the BSS segment (read/write, 0-initialized). |
| .byte N, M, … | Produce a number of bytes, each with one of the specified values. |
| .cpu cortex-m0 | Specify that the target CPU is a Cortex-M0. |
| .data | Following items are produced in the DATA segment (read/write, initialized). |
| .global Name, Name2, … | Make the indicated local labels accessible to other source files. |
| .hword N, M, … | Produce a number of half-words, each with one of the specified values. |
| .word N, M, … | Produce a number of words, each with one of the specified values. |
| .text | Following items are produced in the CODE segment (read-only). |

|  |  |
| --- | --- |
| **Data movement** | |
| MOV Rd, Rm | Copy the content of Rm to Rd |
| MOV Rd, #immed8 | Load the value immed8 into Rd |
| LDR1 Rd, [ Rn, Rm ] | Load Rd with the content of the memory address Rn + Rm |
| LDR1 Rd, [ Rn, #immed52 ] | Load Rd with the content of memory address Rn + immed52 |
| LDR Rd, [ PC3, #immed8 ] | Load Rd with the content of memory address PC + 4 + immed8 \* 4 |
| STR1 Rd, [ Rn, Rm ] | Store the content of Rd in memory at address Rn + Rm |
| STR1 Rd, [ Rn, #immed52 ] | Store the content of Rd in memory at address Rn + immed52 |
| STR Rd, [ PC3, #immed8 ] | Store the content of Rd in memory at address PC + 4 + immed8 \* 4 |
| LDR Rd, =immed32 | Load Rd with the value immed32 |
| 1Append H for a half-word (16-bit) access, B for a byte (8-bit) access.  2The immed5 value in the opcode is multiplied by 4 for a word access, by 2 for a half-word access.  3Can also be SP, which is useful for compiled languages for accessing local variables the stack. | |
| LDRSH Rd, [ Rn, Rm ] | Load Rd with the sign-extended half-word content of the memory address Rn + Rm |
| LDRSB Rd, [ Rn, Rm ] | Load Rd with the sign-extended byte content of the memory address Rn + Rm |
| PUSH { low\_register\_list1 } | Push the indicated registers onto the stack |
| PUSH { low\_register\_list1, LR } | Push the indicated registers and the LR onto the stack |
| POP { low\_register\_list1 } | Pop the indicated registers from the stack |
| POP { low\_register\_list1, PC } | Pop the indicated registers and the PC from the stack |
| 1List of low registers, separated by commas, for instance ”R2, R3, R4, R6”.  A range can be specified, for instance ”R2 – R4, R6”. | |
| LDM Rn, { low\_register\_list } | Load the indicated registers from the memory pointer to by Rn and subsequent memory locations. |
| LDMIA Rn!, { low\_register\_list } | Load the indicated registers from the memory pointer to by Rn and subsequent memory location, then update Rn to point to the next higher memory address. |
| STMIA Rn!, { low\_register\_list } | Save the indicated registers from the memory pointer to by Rn and subsequent memory location, then update Rn to point to the next higher memory address. |

|  |  |
| --- | --- |
| **Arithmetic** | |
| ADD1 Rd, Rn, Rm | Rd = Rn + Rm |
| ADD Rd, Rn, #immed3 | Rd = Rn + immed3 |
| ADD Rd, Rd, #immed8 | Rd = Rd + immed8 |
| ADC Rd, Rn, Rm | Rd = Rn + Rm + Carry |
| SUB Rd, Rn, Rm | Rd = Rn - Rm |
| SUB Rd, Rn, #immed3 | Rd = Rn - immed3 |
| SUB Rd, Rd, #immed8 | Rd = Rd - immed8 |
| SBC Rd, Rn, Rm | Rd = Rn - Rm - Carry |
| MUL Rd, Rn, Rm | Rd = Rn \* Rm 2 |
| 1 For ADD only, all registers are allowed  2 The lower 32 bits of the result are kept, the higher 32 bits are ignored | |

|  |  |
| --- | --- |
| **Logical** | |
| AND Rd, Rn, Rm | Rd = Rn & Rm |
| EOR Rd, Rn, Rm | Rd = Rn | Rm |
| ORR Rd, Rn, Rm | Rd = Rn ^ Rm |
| BIC Rd, Rn, Rm | Rd = Rn & ! Rm |
| MOVN Rd, Rn | Rd = ! Rn |
| All logical operations are bitwise: bit N in the results is affected only by bits N in the source(ses). | |

|  |  |
| --- | --- |
| **Shift** | |
| LSL Rd, Rn, Rm | Rd = Rn << Rm |
| LSL Rd, Rn, #immed5 | Rd = Rn << immed5 |
| LSR Rd, Rn, Rm | Rd = Rn >> Rm |
| LSR Rd, Rn, #immed5 | Rd = Rn >> immed5 |
| ASR Rd, Rn, Rm | Rd = Rn / 2 \*\* Rm |
| ASR Rd, Rn, #immed5 | Rd = Rn / 2 \*\* immed5 |
| ROR Rd, Rn, Rm | Rd = ( Rn >> ( Rm % 32 )) | ( Rn << ( 32 – ( Rm % 32 ))) |

|  |  |
| --- | --- |
| **Extend** | |
| SXTH Rd, Rm | Sign-extend the half-word 2’s-complement value in Rm , store in Rd |
| SXTB Rd, Rm | Sign-extend the byte 2’s-complement value in Rm , store in Rd |
| UXTH Rd, Rm | Zero-extend the half-word unsigned value in Rm , store in Rd |
| UXTB Rd, Rm | Zero-extend the byte unsigned value in Rm , store in Rd |

|  |  |
| --- | --- |
| **Compare and test** | |
| TST Rn, Rm | Calculate Rn & Rm |
| CMP Rn, Rm | Calculate Rn - Rm |
| CMN Rn, Rm | Calculate Rn + Rm |
| CMP Rn, #immed8 | Calculate Rn – immed81 |
| 1range -128 … +127 | |

|  |  |
| --- | --- |
| **Add and subtract SP** | |
| ADD SP, SP, #immed71 | SP = SP + immed7 |
| SUB SP, SP, #immed71 | SP = SP - immed7 |
| ADD Rd, SP, #immed82 | Rd = SP + immed8 |
| 1range 0 ... 512 in steps of 4; 2range 0 ... 1024 in steps of 4 | |

|  |  |
| --- | --- |
| **Endianness conversion** | |
| REV Rd, Rm | Convert a little endian 32-bit value to big endian, or vice versa |
| REV16 Rd, Rm | Convert an unsigned little endian 16-bit value to big endian, or vice versa |
| REVSH Rd, Rm | Convert a signed little endian 16-bit value to big endian, or vice versa, and sign-extend into the higher 16 bits |

|  |  |  |
| --- | --- | --- |
| **Flow control** | | |
| MOV PC, Rm | | Move any register to PC |
| POP { low\_register\_list, PC } | | Pop the indicated registers and the PC from the stack |
| ADD PC, PC, Rm | | Add low register to PC |
| B label | | Branch to the label |
| B*cond*1 label | | Branch to the label if (and only if) the condition holds |
| BX Rm2 | | Branch to the address in the register |
| 1The conditional branch mnemonics can be one from the next table.  2The lowest bit of the register must be 1. | | |
| BL label | Save PC to LR, then branch to the label | |
| BLX Rm1 | Save PC to LR, then branch to the address in the register | |
| 1The lowest bit of the register must be 1. | | |

|  |  |  |
| --- | --- | --- |
| **Instruction** | **Condition** | **Flags** |
| BCC / BCLO | Carry clear / unsigned lower | !C |
| BCS / BHS | Carry set / unsigned higher or same | C |
| BEQ | Equal | Z |
| BGE | Signed greater than or equal | N == V |
| BGT | Signed greater than | !Z & ( N == V ) |
| BHI | Unsigned higher | C & !Z |
| BLE | Signed less than or equal | Z | ( N != V ) |
| BLS | Unsigned lower or same | !C | Z |
| BLT | Signed less than | N != V |
| BMI | Minus / negative | N |
| BNE | Not equal | !Z |
| BPL | Plus / zero or positive | !N |
| BVC | No overflow | !V |
| BVS | Overflow | V |