EL6483: Overview of ARM Cortex-M Processor and Assembly Language Instruction Set

EL6483

Spring 2016
ARM Cortex-M

- 32-bit RISC processor

**Cortex-M4F**

- Cortex-M3 + DSP instructions + floating point unit (FPU) – F in Cortex-M4F stands for FPU
- ARM and Thumb instruction sets (Thumb is a subset of the ARM instruction set encoded as 16-bit instructions)
- includes a set of DSP instructions
- IEEE754-compliant single-precision FPU
- optimized for low power; integrated sleep and deep sleep modes
Registers

- General-purpose registers: R0-R12
- Stack pointer: SP (R13)
- Link register: LR (R14)
- Program counter: PC (R15)
- Special registers: Program status register (PSR), Exception mask registers (PRIMASK, FAULTMASK, BASEPRI), Control register (CONTROL)

Section 2.1.3 of ARM Cortex-M4 Devices Generic User Guide
RISC and CISC processors

- RISC: *Reduced Instruction Set Computing* or *Reduced Instruction Set Computer*
  - e.g., ARM, Atmel AVR, Sun SPARC (now Oracle SPARC), IBM’s Power Architecture

- CISC: *Complex Instruction Set Computing* or *Complex Instruction Set Computer*
  - e.g., x86 (most desktop and laptop processors) from Intel, AMD, etc.

- RISC processors typically utilize simpler instruction sets with each instruction typically executing in a single clock cycle; CISC processors usually provide more *complex* instructions that execute in several clock cycles.
RISC and CISC processors

- RISC processors typically utilize simpler instruction sets with each instruction typically executing in a single clock cycle; CISC processors usually provide more complex instructions that execute in several clock cycles.
  - for example, CISC processors might provide instructions that can directly operate on arbitrary memory locations (e.g., add two numbers specified by their addresses in memory and store the result at another address in memory)
  - RISC processors typically utilize the load/store architecture wherein instructions (except for load and store instructions) operate only on registers. In a RISC processor, adding two numbers in memory locations is done by loading the two numbers into registers, executing an instruction to add the contents of the two registers and store the result into a register, and then storing the result into the desired memory location.
RISC and CISC processors

- RISC instruction sets typically comprise of simple *single-clock* instructions that have been highly optimized (also efficient *pipelining*). Memory is only accessed through *load* and *store* instructions.

- Complex operations implemented in hardware in CISC processors (as complex multi-clock instructions) are instead implemented using software (i.e., multiple lines of assembly language) in RISC processors. While code for a RISC processor is typically bigger than equivalent code for a CISC processor, memory is sufficiently inexpensive (and memory bus speeds are sufficiently high) these days that code size is not *usually* a major constraint.

- RISC processors provide a register-to-register approach in instructions (e.g., *ADD* adds contents of two registers rather than arbitrary memory locations). CISC processors provide a memory-to-memory approach in instructions.

- RISC processors usually have *more general-purpose registers*. 
Cortex-M4 instructions

Some commonly used instructions

- load and store instructions: LDR, STR, etc.
- move instructions: MOV, etc.
- branch (jump) instructions: B, BL, BX, BLX, etc.
- stack push and pop instructions: PUSH, POP, etc.
- Arithmetic operations: ADD, SUB, MUL, SDIV, UDIV, etc.
- No operation (null operation): NOP

Section 3.1 of ARM Cortex-M4 Devices Generic User Guide
Section 3.3.1 of ARM Cortex-M4 Processor Technical Reference Manual
Exceptions/Interrupts

- interrupt the “normal” flow of the program to handle various types of events as and when they occur
- commonly used programming model in embedded systems ... low latency response to various hardware/software generated events (e.g., when a byte arrives on a serial port, when an analog-to-digital conversion is completed, when transmission of a byte on a serial port is completed, etc.)
  - the main loop (normal flow of the program) can simply be a

```c
while (1){}
```

- exception types also include reset, various types of faults (e.g., memory protection fault, memory bus fault, hard fault), system events (e.g., non maskable interrupt, system tick, supervisor call)
ARM Cortex-M4 Exceptions/Interrupts

- up to 255 interrupt handlers
- first 10 interrupt sources are defined by ARM; the other interrupt sources can be defined for specific processor chips by the processor vendors (e.g., ST, TI) and could be different between different specific Cortex-M4 based processors
- interrupt handlers defined in a vector table located, by default, in memory segment starting from 0x00... vector table is a list of addresses of functions/labels that will handle the different types of interrupts/exceptions. These functions (interrupt service routines) can be implemented in C/C++/Assembly.
- first entry on the vector table is the initial value of the stack pointer

Section 2.3 of ARM Cortex-M4 Devices Generic User Guide
Processor modes in ARM Cortex-M4:

- **Thread mode:** normal mode used to execute application software; processor enters thread mode when it comes out of reset
- **Handler mode:** mode used to run interrupt handlers; processor returns to thread mode after finishing handling the exceptions

- On some other ARM architectures, modes include User mode, FIQ (fast interrupt request), IRQ (interrupt request), Supervisor, Abort, Undefined, System, and Monitor.

Section 2.1.1 of ARM Cortex-M4 Devices Generic User Guide
Each exception/interrupt line can be in one of multiple states at a specific time:

- **Active**: the exception is currently being serviced (i.e., its interrupt service routine is running).
- **Pending**: the exception is waiting to be serviced (e.g., a byte has arrived from a serial port, but the corresponding interrupt service routine has not yet been run since another interrupt service routine is running).
- **Active and pending**: the exception is currently being serviced and there is another pending exception of the same type.
- **Inactive**: the exception is not active and not pending.
In general, an exception can interrupt the running of another interrupt service routine (then, two exceptions are in the active state at the same time). To prevent interrupting of an interrupt service routine, *masking* can be used (then, new exceptions will have to wait for the interrupt service routine to complete).

However, it is not good to *mask* interrupts for too long since events (e.g., bytes from a serial port, analog-to-digital conversion updates) can be possibly missed. A typical embedded system strategy to reduce the *masking* time period is to make the interrupt service routines very short (e.g., only read the new data from the external hardware and set a flag) and have the main loop continuously poll for *flag* variables that are set in the interrupt service routines.
Cortex-M4 instruction set

- Some commonly used instructions:
  - load and store instructions: LDR, STR, etc.
  - move instructions: MOV, etc.
  - branch (jump) instructions: B, BL, BX, BLX, etc.
  - stack push and pop instructions: PUSH, POP, etc.
  - Arithmetic operations: ADD, SUB, MUL, SDIV, UDIV, etc.
  - No operation (null operation): NOP

The Cortex-M4 instruction set is listed in Section 3.1 of ARM Cortex-M4 Devices Generic User Guide. Also, section 3.3.1 of ARM Cortex-M4 Processor Technical Reference Manual.
Load and store instructions

- **LDR**, **STR**: “Load register” and “Store register”

Examples:

- **LDR R1, [R0]**; load into R1 the content of the memory location whose address is in R0
- **STR R1, [R0]**; store the contents of R1 into the memory location whose address is in R0
- **LDR R1, [R0,#4]**; #4 specifies an offset value, load into R1 the content of the memory location whose address is given by the value R0 + 4
- **LDR R1, [R0,#4]!**; #4 specifies an offset value, increment R0 by 4, load into R1 the content of the memory location whose address is given by the new contents of R0; with ! at the end, the value of R0 is also changed
Move operations

- MOV, MVN: “Move” and “Move NOT”; move not does a bitwise logical NOT operation before copying the value to the destination register.

Examples:

- MOV R0, R1; copy the contents of R1 to R0
- MOV R0, #10; set R0 to 10
- MVN R0, R1; set R0 to bitwise NOT of contents of R1
- MVN R0, #1; set R0 to 0xFFFFFFFFFE (bitwise NOT of 0x1)
Branch operations

- B, BL, BX, BLX

- B and BL are branch with “immediate” arguments (e.g., jumping to a label/function); the symbol . is a synonym for the current program location.

- BX and BLX are branch with “register” arguments (i.e., jump to an address stored in a register).

- BL and BLX store a bookmark to the *current place in the program* by writing the address of the next instruction to the link register (LR).
Branch operations

- **B, BL, BX, BLX**

- **Examples:**
  - `B labelA`; branch to the label `labelA`
  - `BL labelA`; update LR and branch to the label `labelA`
  - `BX LR`; branch to the location whose address is in LR, e.g., return from a function call
  - `BLX R0`; update LR and branch to the location whose address is in R0
  - `B .`; branch to the current program location (infinite loop)

- **LR** (link register – register R14) is used to store a bookmark to the current location in the program, e.g., to store the return information for function calls.

- **PC** (program counter – register R15) contains the current program address; branch instructions update the program counter.
Push and pop operations (Stack)

- **PUSH, POP**
  
  - **Examples:**
    - `PUSH {R1}`  ; push the contents of R1 onto the stack
    - `PUSH {R0,R1}` ; push the contents of R0 and R1 onto the stack
    - `PUSH {R0,R2-R4}` ; push the contents of R0, R2, R3, R4 onto the stack
    - `PUSH {R0,LR}`  ; push the contents of R0, LR onto the stack
    - `POP {R0,R1}`   ; pop the top two 32-bit values from the stack into R0 and R1
  
  - **PUSH and POP operations update the stack pointer (SP). The SP points to the top of the stack.**
Arithmetic instructions

Addition and subtraction

- ADD, SUB: Add and subtract
- ADC, SBC: "Add with carry" and "Subtract with carry"
- The carry instructions also utilize the carry flag (set by previous instructions - the carry flag is stored as a bit in the application program status register)

Examples:

- ADD R1, R0, R1 ; set R1 to the sum of the contents of R0 and R1
- SUB R2, R0, R1 ; set R2 to the difference of the contents of R0 and R1
- ADD R2, R0, #10 ; R2 = R0 + 10
- first argument to the instruction is the destination register; the next two arguments are the two operands (the quantities on which the instruction operates).

- RSB: reverse subtract (i.e., subtract the contents of the second operand from the third operand), e.g.,
  RSB R2, R0, R1 ; set R2 to the difference of the contents of R1 and R0
- Signed and unsigned variants: SADD16, etc.
Multiplication and division

- **MUL, MLA, and MLS**: “Multiply”, “Multiply with accumulate”, and “Multiply with subtract”
- **Examples**:
  - `MUL R2, R0, R1` ; `R2 = R0*R1`
  - `MLA R3, R0, R1, R2` ; `R3 = R0*R1 + R2`
  - `MLS R3, R0, R1, R2` ; `R3 = R0*R1 - R2`
- Signed and unsigned variants: SMLA, etc.
- **SDIV, UDIV**: “Signed division” and “Unsigned division”
- **Examples**:
  - `SDIV R2, R0, R1` ; signed divide, `R2 = R0/R1`
  - `UDIV R2, R0, R1` ; unsigned divide, `R2 = R0/R1`
The basic syntax of many of the instructions is:

```
op{S}{cond} Rd, Rn, Operand2
```

- **op**: a 3-letter mnemonic for the operation (e.g., ADD)
- **S**: optional suffix; if this suffix is used, it indicates that the condition code flags should be updated as a result of the operation
- **cond**: optional suffix indicating a condition code; this specifies that the operation should only be performed if the condition is satisfied; Examples: EQ for “if equal”, GT for “greater than”; the condition is tested based on condition code flags set by a previous instruction
- **Rd**: destination register
- **Rn**: register holding the first operand for the instruction
- **Operand2**: flexible second operand, which can be a constant or a register (with optional shift); a constant is specified as, for example, #5.
Floating point instructions

- Floating point instructions are available if there is a floating point unit (FPU) in the system and is enabled (the FPU is generally enabled as part of the start-up sequence on ARM Cortex-M4F).
- Examples of floating point instructions
  - VADD, VSUB, etc.: floating point addition and subtraction
  - VMUL, VDIV, etc.: floating point multiplication and division
  - VABS: floating point absolute value
- The FPU has 32 singleword (32-bit) floating point registers S0, ..., S31. The floating point instructions have similar syntax to the regular instructions but with a prefix of V (e.g., VMOV, VADD, VPUSH, VPOP, etc.). Since many of the floating-point instructions can handle either 32-bit (singleword) operands or 64-bit (doubleword) operands (which have the corresponding registers D0, ..., D15), the instructions typically take a suffix such as .F32 or .F64 to indicate the size of the operands. Examples: VMOV.F32, VADD.F32, etc.
Bitwise operations

- **AND, ORR (logical OR), etc.;** exclusive OR is **EOR**
- **LSL, LSR:** “logical shift left” and “logical shift right”
- **ROR:** rotate right

**Examples:**

- `AND R2, R0, R1 ; R2 = R0 & R1`
- `ORR R2, R0, R1 ; R2 = R0 | R1`
- `AND R2, R0, #0x10 ; R2 = R0 & 0x10`
- `ORR R2, R0, #0x10 ; R2 = R0 | 0x10`
- `LSL R2, R0, #2 ; R2 = R0 << 2`
Compare operations

- **CMP, CMN**: “Compare” and “Compare negative”
  - CMP is based on subtracting the second operand from the first operand; CMN is based on adding the second operand to the first operand.
  - The compare operations do not change the contents of the operand registers.
  - Examples:
    - `CMP R0, R1`; Compare contents of R0 and R1
    - `CMP R0, #5`; Compare contents of R0 with 5
  - CMP and CMN update *condition flags*; condition flags can also be updated by many of the other instructions (e.g., addition).
Condition code flags

- The application program status register is a special register that contains various flags that are updated during program execution including the following condition code flags:
  - $N$: set to 1 when the result of an operation was negative, cleared to 0 otherwise
  - $Z$: set to 1 when the result of an operation was zero, cleared to 0 otherwise
  - $C$: set to 1 when the operation resulted in a carry, cleared to 0 otherwise
  - $V$: set to 1 when the operation caused an overflow, cleared to 0 otherwise

- Many instructions provide an optional $S$ suffix to specify if the condition code flags must be updated, e.g., ADDS, SUBS, MOVS, LSLS, etc., update the condition code flags based on the result of the operation.
Conditional execution of instructions

- Many instructions support optional suffixes denoting various conditions to specify that the instruction must be executed only if the specified condition is true.

- Some examples of condition suffixes:
  - EQ for equal (i.e., \( Z = 1 \)), NE for not equal (i.e., \( Z = 0 \))
  - GT for “greater than” and LT for “less than”; these conditions are evaluated using combinations of flags \( Z, N, \) and \( V \). For example, GT is equivalent to “\( Z=0 \) and \( N = V \)”, LT is equivalent to “\( N \neq V \)”
  - GE and LE for “greater than or equal” and “less than or equal”

- Example: ADDEQ will do an addition only if the “equal” condition is currently active (i.e., a previous instruction caused the \( Z \) flag to become 1); BNE will do a branch (jump) only if the “equal” condition is not currently active.
Function definition in Assembly

- Assembly language functions (subroutines) are defined in a .s file.
- The exact format of the Assembly file depends on the specific Assembler.
- For the GNU assembler (arm-none-eabi-as), put a line like the following at the beginning of the file:
  .section .text
- Define a function using a label such as `func`:
- Use the `.global` keyword to make the function visible from other files (i.e., other translation units).
- Example:

```assembly
.section .text
.syntax unified
.global func
func:
    ADD R0,R0,#1    /* Add 1 to R0 */
    BX LR           /* Return from the function (subroutine) */
```
Function definition in Assembly

- Example:

```
.section .text
.syntax unified
.global func
func:
  ADD R0,R0,#1 /* Add 1 to R0 */
  BX LR       /* Return from the function (subroutine) */
```

- By convention, all lines are indented with some space except for label names (function names are simply label names – more generally, labels can be used to implement equivalents of C for loops and while loops, etc.).

- Put .syntax unified to use a more standardized unified syntax.

- Comments are indicated using /* ... */ as in C.
Sections in binary executables

- `.text`: the executable code
- `.data`: initialized data
- `.bss`: uninitialized data; also, data that is zero
- `.rodata`: read-only (const) data
- other sections might be defined as well (e.g., see `stm32_flash.ld`)
To call another function (Assembly/C), use BL. Since the LR will get overwritten, save LR before calling BL (for example, save LR on stack).

```
.section .text
.syntax unified
.global func1, func2

func1:
PUSH {LR}  /* save LR (R14) on stack */
BL func2  /* call function func2 */
POP {LR}  /* restore LR (R14) from stack */
BX LR     /* Return from the function */

func2:
ADD R0,R0,#1
BX LR     /* Return from the function */
```
To access the address of a symbol (which could be defined in C or Assembly), use `%`. For example, if a variable `b` is defined in C as the global variable `int b`, the memory address of this variable can be copied to R0 using:

```
LDR R0, =b
```

To copy the value of `b` into the register R1 for example, use:

```
LDR R0, =b
LDR R1, [R0]
```

To call a function `g` defined in C, use:

```
LDR R0, =g
BLX R0
```
Function definition in Assembly

- For an example of a .s file, see
  STM32F4-Discovery_FW_V1.1.0/Libraries/CMSIS/ST/STM32F4xx
  /Source/Templates/TrueSTUDIO/startup_stm32f4xx.s
  This folder path is included in STM32_example.zip from Homework 1 for example.

- Another way to see examples of Assembly code is to use
  arm-none-eabi-objdump to disassemble a binary executable (ELF file) and see the Assembly code generated for corresponding C code. For example:

  arm-none-eabi-objdump -S STM32_example.elf
  To see Assembly code corresponding to a function f for example, look for a section in the output of the arm-none-eabi-objdump command labeled as <f>. The output of this arm-none-eabi-objdump command will also include memory addresses (e.g., 8002dfc:) and machine code (e.g., b538).
  When running code in the debugger (arm-none-eabi-gdb), you can see the corresponding compiled Assembly using the following command at the (gdb) prompt:
  disassemble /m
Weak visibility: .weak

- *Weak* visibility is used to be able to provide a default definition of a symbol that can be overridden elsewhere. This visibility attribute is often used in startup code (e.g., `startup_stm32f4xx.s`) to provide default handlers for various interrupts. But, if functions for any interrupt handlers are defined in user code, then these user-defined functions will take precedence over the default handlers.

- In the absence of the weak visibility mechanism, if the same symbol name is defined in multiple files, we will have a link-time error. With the *weak* visibility attribute, the weak symbol definition will be hidden if a normal symbol definition is provided in another file.

- The weak visibility method is also used in defining default functions in software libraries (e.g., to define a default function to process data from a sensor) that can be overridden in user code.
Weak visibility: .weak

- Example of defining a weakly visible function in Assembly:
  
  ```assembly
  .weak func1

  func1:
      BX LR
  ```

- Example of defining a weakly visible function in C:
  
  ```c
  int func1(int) __attribute__((weak));

  int func1(int a)
  {
  }
  ```
Assembly code can be included directly in a C file, i.e., *inline* Assembly.

Examples:

```c
asm("ADD R0,R0,#2");
asm("MOV R1,R0 \n\t" 
    "ADD R0,R1,#2\n\t" 
    "ADD R0,R0,#3" 
);
```

In some compilers, the keyword `__asm` or the keyword `__asm__` is used for inline Assembly. The syntax above will work with GCC.
ARM function calling convention (both C and Assembly)

- Registers R0 to R3 are utilized to pass argument values to a function and also to hold results returned from a function.
- Register R12 is an intra-procedure-call scratch register that can be freely modified within a function.
- Registers R4 to R11 can be used to hold local variables.
- Register R13, R14, R15
  - R13 is the stack pointer.
  - R14 is the link register.
  - R15 is the program counter.
- If the arguments to a function cannot fit within registers R0 - R3, then the remainder of the arguments are placed on the stack before making the function call.
- Return value of the function is returned in R0 - R3. If the result of the function cannot fit within R0 - R3, then the calling function (caller) allocates space for the return value and passes the address to that memory space in register R0.
ARM function calling convention: *Caller* and *Callee*

- *Caller* is the function making the function call; *callee* is the function that is called (callee may be a caller for some other function).

- *Callee* functions may modify R0 - R3, R12, and R14.
  - Callee functions are not allowed to modify other registers (i.e., the values of the other registers must be the same after the function returns as when the function was called; if the function needs to use these registers, it can save them to stack and restore them before returning).
ARM function calling convention: Caller and Callee

- Hence, the typical sequence of steps in a callee function is:
  - Push registers R4 - R11 (or the subset of these registers that are modified in the function) and register R14 onto the stack.
  - Do calculations using the passed arguments (in R0 - R3 and additional arguments on stack if necessary) and scratch registers R4 - R12.
  - When required, call other functions (using BL or BLX) assuming that registers R0 - R3, R12, and R14 may be modified by the function call.
  - Put the return value into R0 (or R0 - R3 if required and/or into memory space pointed to by R0 if required).
  - Pop registers R4 - R11 and R15 from the stack. The value of register R14 that was pushed on the stack in the beginning of the function is popped into R15 (program counter); this effectively results in a function return (this would be equivalent to popping R14 and then doing a BX LR).
ARM function calling convention: *Caller* and *Callee*

- The link register (R14) stores a bookmark to the calling function to be able to return to the correct place from a function call. Since the callee function may modify the link register, the *caller* function must save the link register value (i.e., the bookmark to the return place in *its* calling function) before making a function call.

- Registers caller needs to save (if required): R0 - R3, R12, R14

- Registers callee guarantees will be saved/restore: R4 - R11, R13

- Multiple registers can be pushed to stack using a single PUSH instruction and restored using a single POP instruction, e.g., PUSH {R0-R3,R12,R14}

- Callee also guarantees that register R15 (program counter) will be changed to point to the next instruction when the callee function returns.
The function-calling convention for floating-point arguments is similar to integer arguments except that the floating point registers are used instead.

For example, if a function takes two float parameters, these parameters are passed in registers S0 and S1; if a function returns a float, it is returned in register S0.

Example of an Assembly function that can be called from C using the function prototype `float mul2(float a, float b);` and will return the value of `a*b`:

```assembly
mul2:
    VMUL.F32 S0,S0,S1
    BX LR
```

When called, for example, as `mul2(1.1, 2.2)`, this function will return 2.42.

Note that you can use the command "info all-registers" at the `(gdb)` prompt of `arm-none-eabi-gdb` to see the values of the floating-point registers.