UNIVERSITÉ DE NANTES

LS2N LABORATOIRE DES SCIENCES DU NUMÉRIQUE DE NANTES

Embedded Software Systems

Lecture: Mixing C and Assembly

Sébastien Faucou
firstname.name@ls2n.fr

Université de Nantes
LS2N, UMR CNRS 6004
Real-Time Systems group

Lecture overview and motivations

Lecture overview

- Instruction Set Architecture (ISA)
- Mapping C to Machine Language
- Application Binary Interface

Motivations

When programming embedded software systems, it is not always possible to ignore what is going on at the lower abstraction levels

- Performances and energy efficiency depends on a good usage of hardware resources
- Some software predictability issues often have their roots at lower levels
- Some hardware resources might not be directly available through high level languages API

What you should know, what you will learn

To follow this lecture you should:

- Have a basic knowledge of computer architecture
- Have a good understanding of data representation
- Know how to program in C (intermediate level)

We will improve our knowledge and skills wrt. the following topics:

- How to integrate C and Assembly
- The ARMv7-M Instruction Set Architecture

These are prerequisites for the next lecture on RTOS internals.

References

David A. Patterson and John L. Hennessy, *Computer Organization and Design - The Hardware/Software Interface - ARM Edition*, Morgan Kaufmann, 2017.

- A classical textbook to introduce computer architecture. Topics in this lecture are studied in chapter 2.

Joseph Yiu, *The definitive guide to ARM Cortex-M3 and Cortex-M4 processors - Third Edition*, Newnes, 2014.

- A reference document providing complete low level technical details.

Links to reference document from ARM are also provided in the slides.

Part I

Instruction Set Architecture

The concept of Instruction Set Architecture

Instruction Set Architecture (ISA)

The *Instruction Set Architecture (ISA)* of a machine is a programming interface for the hardware/software boundary. It is directly used for the development of software components such as compilers, operating systems, or device drivers. It is also the target of the compiler when translating high-level programs to machine-level programs.
A machine implements at least one ISA.

Not to be confused with

Microarchitecture

The *microarchitecture* of a machine is the actual hardware organization of the machine that implements one (or more) ISA.

The components of an ISA

An ISA is usually composed of

An *instruction set*	The "vocabulary" of the machine
A *register bank*	A small set of fast temporary data storage locations to perform arithmetic and logic operations and control the execution of the program
A *memory system*	A set of slower locations to store programs and their data

The components of the ISA are an abstraction of the components of the microarchitecture.

The ARMv7-M ISA

The ARM-v7M ISA is used by ARM CORTEX M-3 and M-4 families.
It consists in :

- 16 registers used for application programming
- A set of special registers
- A memory system with 32 bits wide addresses
- A reduced set of instructions

Reference documents

Full Instruction Set specification:

`http://infocenter.arm.com/help/topic/com.arm.doc.dui0553b/DUI0553.pdf`

Quick Reference card:

`http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001l/QRC0001_UAL.pdf`

Instructions

Instructions, Instruction Set

Instruction

An instruction is composed of an *operation code* (opcode) and zero or more *arguments*. Arguments can be registers, memory locations, or immediate values (constants).

An instruction has two representations:

- A textual representation for developers
- A binary representation for machines

The tool used to go from textual to binary is an *assembler*.
The tool used to go from binary to textual is a *disassembler*.

$\rightsquigarrow$ *Illustration with file add.c*

Some usefull instructions

Mnemonic	Description
LDR	Load reg. from memory
STR	Store reg. to memory
POP	Pop stack top to reg.
PUSH	Push reg. to stack
MOV	Move between reg.
LSL	Logical shift left reg.
ADD	Addition

Mnemonic	Description
SUB	Subtraction
CMP	Compare
TST	Test
B	Branch
BL	Branch with Link
BX	Branch indirect
CBZ	Compare and br. if zero

Registers

Register bank

Register bank

A register is a small set of fast locations to store data inside the processor.

Arithmetic and logic instructions can only update registers
So, to compute with data located in the memory system, 3 steps are required:

- load data into a register
- perform a bunch of operations on the data
- write back result to the memory system

Without optimization, a compiler will apply this approach for each operations. An optimizing compiler will minimize the number of load/store operations (why ?)

$\rightsquigarrow$ *Illustration with file tabsum.c*

Main registers of the ARMv7-M

Register	Possible names	Notes
R0 to R12	R0, R1 ... R12, r0, r1, ... r12	general purpose registers
R13	R13, r13, SP, sp	stack pointer
R14	R14, r14, LR, lr	link register
R15	R15, r15, PC, pc	program counter

Table: Register bank of the ARMV7-M ISA

Beside the register bank, there exist a set of *special registers* used to store the processor state, handle exceptions, control the status of interrupts, or the execution mode.

PSR: program status register

The *Program Status Register* (PSR) is a special register used to store the processor state. It is split into Application PSR (APSR), Execution PSR (EPSR) and Interrupt PSR (IPSR).

APSR is a *bitfield* to store information on the result of instructions:

Flag	aka	Description
N	Negative	Signed interpretation of result is negative
Z	Zero	All bits of result are 0
C	Carry	Carry bit is not null
V	oVerflow	Carry bit and most significant bit differ

Table: APSR Integer status flags

APSR bits are set:

- by explicit instructions such as `CMP` or `TST`
- by arithmetic and logic instructions (and some others) when suffix `S` is used (eg. `ADDS`, `SUBS`)

APSR Integer status flags update examples

Operation	Result	Flags
0x70000000 + 0x70000000	0xE0000000	N=1, Z=0, C=0, V=1
0x90000000 + 0x90000000	0x30000000	N=0, Z=0, C=1, V=1
0x80000000 + 0x80000000	0x00000000	N=0, Z=1, C=1, V=1
0x00001234 - 0x00001000	0x00000234	N=0, Z=0, C=1, V=0
0x00000004 - 0x00000005	0xFFFFFFFF	N=1, Z=0, C=0, V=0

APSR and conditional branching

APSR bits are used to drive *conditional instructions*. An opcode suffix is used to select the condition:

Suffix	Condition	APSR
EQ	Equal	Z = 1
NE	Not equal	Z = 0
GE	Signed greater or equal	N=V
GT	Signed greater than	Z=0 & N=V
LE	Signed lower or equal	~(Z=0 & N=V)
LT	Signed lower than	N = ~V
HS	Unsigned higher or same	C=1
HI	Unsigned higher	Z=0 & C=1
LS	Unsigned lower or same	~(Z=0 & C=1)
LO	Unsigned lower	C=0

Table: Some condition suffix

⇝ *Illustration with file tabsum.c*

Addressing the memory system

Adress space

The memory system of armv7-m uses a single flat *address space* of 2^{32} bytes.

Most accesses will be *word aligned* or *half-word aligned*.

- Word addresses are divisible by 4. Word at address A are the 4 bytes at address A, A+1, A+2 and A+3
- Half-word addresses are divisible by 2. Half-word at address A are the 2 bytes at address A and A+1

Usually, embedded systems do not include a virtual memory system and their physical memory is a few MB, so most addresses will not be mapped to actual locations (access will result in hardware exception).

Overview of system address map

The address space is divided in 8 partitions of 0.5 GB. In practice, most adresses of each partition is not mapped to an actual location.

Here are the partitions for on-chip resources:

Address range	Section	Description
0x00000000 - 0x1FFFFFFF	Code	Typically ROM or Flash
0x20000000 - 0x3FFFFFFF	SRAM	On-chip RAM
0x40000000 - 0x5FFFFFFF	Device	On-chip Devices

Table: System memory map for on-chip resources

Memory access operations

`LDR` allows to load a memory data in a register.
`STR` allows to store part of a register to memory

A suffix can be added to the instruction to specify the size of the data.

Suffix	Size
`B`	Byte
`SB`	Signed byte (only for `LDR`)
`H`	Half-word
`SH`	Signed Half-word (only for `LDR`)

Table: Size suffix for memory accesses

`PUSH` to push a register on top of the stack
`POP` to pop the top of the stack to a register

Addressing the memory

The address for a load or store is formed of:

- A value taken in a *base register*
- An *offset*

The offset can be:

- an immediate (constant) `LDR R0, [R1, #4]`
- a register (not PC) `STRB R0, [R1, -R2]`
- a scaled register (not PC) `LDRH R2, [R5, R6, LSL #2]`

Pre-indexed, post-indexed

Given a base register and an offset, 3 address computation modes are provided:

- *Offset*: add/sub offset to base value

  ```
  LDR R0, [R1, #8]      /* R0 <- M[R1+8] */
  ```

- *Pre-indexed*: add/sub offset to base value, then base is set to this new address

  ```
  LDR R0, [R1, #8]!     /* R0 <- M[R1+8], R1 <- R1+8 */
  ```

- Post-indexed: get value from base only, then set base to the sum/sub of base and offset

  ```
  LDR R0, [R1], #8      /* R0 <- M[R1], R1 <- R1+8 */
  ```

Part II

Application Binary Interface

The concept of Application Binary Interface

Application Binary Interface

An Application Binary Interface (ABI) defines how data structure and procedure are accessed in machine code in order to execute in a specific execution environment.

It is a machine-code level concept.

Not to be confused with

Application Programming Interface

An Application Programming Interface (API) is a set of clearly defined communication methods between software components.

It is a source-code level concept.

The ARMv7-M ABI

Reference document

Overview of all ABI for ARM architecture:

`http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036b/IHI0036B_bsabi.pdf`

Procedure call standard for the ARM architecture:

`http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042f/IHI0042F_aapcs.pdf`

We will review:

- Machine level representation of C data types
- Procedure call

Endianness

Endianness

In machine-level data representation, the *endianness* is the mapping of multi-byte object in words:

- In the *little-endian* view, the least significant byte is at the lowest address
- In the *big-endian* view, the least significant byte is at the highest addressœ

Gnu toolchain allows to generate machine code for both big-endian and little-endian ARM machine. Default is little-endian.

Data alignement

Alignement

Address a is n aligned iff $a \mod 2^n = 0$.
ABI usually define data structure alignement constraints to ensure a correct execution of operations.

Here are the constraints defined in AAPCS for integral and pointer types:

Machine type	Byte size	Byte alignement
Unsigned byte	1	1
Signed byte	1	1
Unsigned half-word	2	2
Signed half-word	2	2
Unsigned word	4	4
Signed word	4	4
Unsigned double word	8	8
Signed double word	8	8
Data pointer	4	4
Code pointer	4	4

Representation of C integral scalar types

C type	Machine type
`char`	unsigned byte
`unsigned char`	unsigned byte
`signed char`	signed byte
`_Bool / bool`	unsigned byte
`[signed] short`	signed halfword
`short`	halfword
`[signed] int`	signed word
`int`	word
`[signed] long`	signed word
`long`	word
`[signed] long long`	signed double-word
`long long`	double-word
`T *`	data pointer
`T (*F)()`	code pointer

Representation of composite types

Aggregates (struct)

- alignement: alignement of the most-aligned component
- size: smallest multiple of the alignment sufficient to hold all members

$\rightsquigarrow$ *Illustration with file struct.c*

Arrays

- alignement: alignement of base type
- size: size of base type $\times$ number of elements

Procedure call: requirements

When calling a procedure, the following requirements should be met:

- The caller should provide parameter values to the callee
- The caller should transfer control to the callee
- The callee should provide a result to the caller
- Execution of the callee should not alter the state of the caller except for:
  - Objects the address of which have been provided by the caller (through pointers)
  - The result
- The callee should transfer control back to the call point (since it can be called from different points in the program)

Simple case: leaf procedure, few parameters

A leaf procedure is a procedure that does not emit procedure call.

$\rightsquigarrow$ *Illustration with files simple_caller.c and simple_callee.c*

- how are parameters passed from the caller to the callee?
- which instruction is used to transfer control from the caller to the callee?
- how is the result passed from the callee to the caller?
- how is the control transfered to the point of call?

A caller callee

When a called procedure becomes the callee of a nested call, things become a bit more complicated.

$\rightsquigarrow$ *Illustration with files callee2.c*

- what is the purpose of operations `push` and `pop`?
- why is it needed in this case (and not in the previous one)?

The stack

Stack

In computer science, a *stack* is a Last In First Out (LIFO) data structure.

In machine-level programming, the stack is the zone of the memory associated to a running program that is used to handle procedure call.

Each call to a procedure pushes a *call frame* on the stack to backup registers and, if needed, store local variables. The frame is popped when the call returns to the caller.

As call frames enter and leave the zone in LIFO order, the zone is named stack.

To ease the management of the stack, ISA usually provide:

- a register dedicated to store the address of the top of the stack: `stack pointer`
- operation to push/pop data

The stack in the ARMv7-M ISA

- As in many ISA, the stack grows toward lower addresses (address of the top is smaller than address of the base)
- Stack pointer is `r13` (other names: `sp`, `psp`)
- Operations `push` and `pop` are provided to ease stack management
  Both take as argument a list of registers (pushed from left to right, popped the other way)
- At every instant, `SP mod 4 = 0` must hold
- At *public interface*, `SP mod 8 = 0` must hold

An example with long arguments and results

Four registers are used for passing arguments, what happens when it is not enough?

$\rightsquigarrow$ *Illustration with files longargs.c*

- what happens when the size of the arguments is too big to fit in `r0-r3`?
- what happens when the size of a data is bigger than a register?

Registers and procedure call

Reg	aka	Role in procedure call
r0, r1		argument / result / scratch register
r2, r3		argument / scratch register
r4 to r8		variable registers 1 to 5
r9	fp	platform register or variable register 6
r10, r11		variable registers 7 and 8
r12	ip	intra-procedure-call scratch register
r13	sp	stack pointer
r14	lr	link register
r15	pc	program counter

Any register in r4-r15 used by the callee has to be restored at the end of the procedure.

Eight registers can be used to store local variables. What to do when it is not enough?

Summary of the procedure call

Caller

1. stores arguments from left to right to `r0-r3` and then the stack
2. calls the procedure with `bl procname`
8. get the result from `r0-r1`

Callee

3. push on the stack all registers in `r4-r12,r14` used in the code of the procedure
4. compute and store the result in `r0-r1`
5. restore modified registers from the stack
6. returns to the call point with `b lr`

Calling Assembly from C

With the GNU toolchain, it is possible to integrate assembly source code in the build process.
It is also possible to generate assembly source code from C code.

To integrate a function given as assembly code:

1. write a C file containing an empty function with the correct prototype
2. generate the assembly code of this function
3. use this file as a starting point (and delete the original C file)
4. make sure to respect the ABI, especially the stack constraints at public points

Calling C from Assembly

Follow rigorously the procedure call mechanisms explained above.

Use the name of the function as the target of the call (eg. `bl funcname`): the linker will insert the expected address during the symbol resolution step.

Part III

Exercises

The value *0xDEADBEEF* is stored in memory at adress `0xFFFF1000`.

Draw a byte-level representation of memory in the range `0xFFFF1000`-`0xFFFF1004` in big and little endian byte ordering.

Let the C type definition below:

```
struct list {
    struct cell* next;
    char b;
};
```

- What is its alignment?
- What is its size?
- Draw a representation of a `struct list` data in memory.

Let the C code below:

```
if (a < b) {
    result = b - a;
} else {
    result = a - b;
}
```

Let us assume that `a` is mapped to `r4`, `b` is mapped to `r1` and `result` is mapped to `r6`. Write a correspoding program in ARMv7-M machine code.

Let the C function with the prototype below:

```
int32_t g(int32_t, int32_t);
```

Give the armv7-m code to call `g(a,b)`, assuming that `a` is allocated to `r4` and `b` is allocated to `r5`.

Considering the same prototype for g as in the previous exercise, provide an armv7-m code corresponding to the following function:

```
int32_t f(int32_t a, int32_t b, int32_t c, int32_t d)
{
    return g(g(a,b), c+d);
}
```

Give a C code corresponding to the function below:

```
00000000 <swap>:
   0:        008b                lsls        r3, r1, #2
   2:        3304                adds        r3, #4
   4:        58c2                ldr        r2, [r0, r3]
   6:        f840 2021           str.w        r2, [r0, r1, lsl #2]
   a:        50c2                str        r2, [r0, r3]
   c:        4770                bx        lr
   e:        bf00                nop
```

Let the C code of function `bubblesort` and the corresponding ARMv7-M machine code.

```
void bubblesort (int32_t v[], int32_t len) {
    int32_t i, j = len, s = 1;
    while (s) {
        s = 0;
        for (i = 1; i < j; i++) {
            if (v[i] < v[i - 1]) {
                swap(v, i-1);
                s = 1;
            }
        }
        j--;
    }
}
```

1. Describe the mapping of variables to registers
2. Execute by hand the machine code. Assume that `v = { 3, 2, 1, 1 };` and `len = 4`.