

- ▶ The kernel operates at the **hardware/software interface**; to frame an investigation, we (ideally) need a reference hardware platform.

- ▶ The kernel operates at the **hardware/software interface**; to frame an investigation, we (ideally) need a reference hardware platform.
- ▶ **Question:** which one?
- ▶ **Answer:** among many viable options, we'll select

**ARM® = arm**

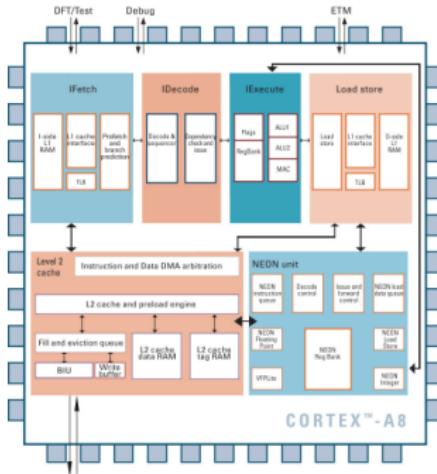
motivated, for example, by

- ▶ (relative) simplicity,
- ▶ ubiquity, and
- ▶ reusability of skills acquired.

- ▶ The kernel operates at the **hardware/software interface**; to frame an investigation, we (ideally) need a reference hardware platform.
- ▶ **Goal:** brief, high-level overview of ARM-based
  - 1. processor design and capabilities, plus
  - 2. assembly language programming.

- ▶ In fact, saying “ARM” is imprecise: it can mean
  1. an ISA:  $\text{ARMv}x \Rightarrow \text{ARM architecture version } x \simeq \text{ISA version } x$ , or
  2. a processor:  $\text{ARM}x$  ( $x \in \{1, 2, \dots, 11\}$ ), Cortex-A/R/Mx and SCx00.

- We'll focus on the 32-bit RISC(ish) **Cortex-A8** [7] processor

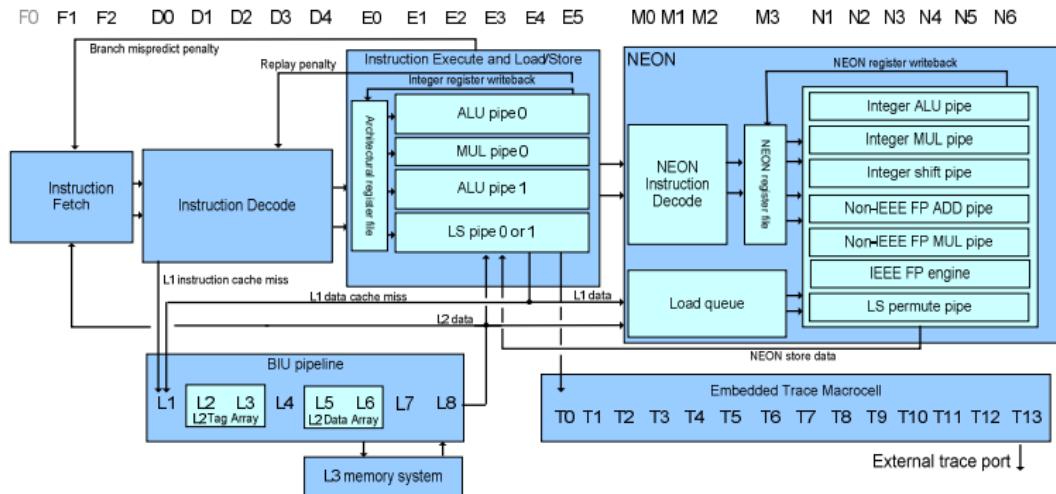


which implements the **ARMv7-A** [6] ISA ...

1. it's a **register machine**, and
2. it's a **load-store architecture**.

[http://www.arm.com/files/pdf/ARM\\_Arch\\_A8.pdf](http://www.arm.com/files/pdf/ARM_Arch_A8.pdf)

- We'll focus on the 32-bit RISC(ish) Cortex-A8 [7] processor



which implements the ARMv7-A [6] ISA ...

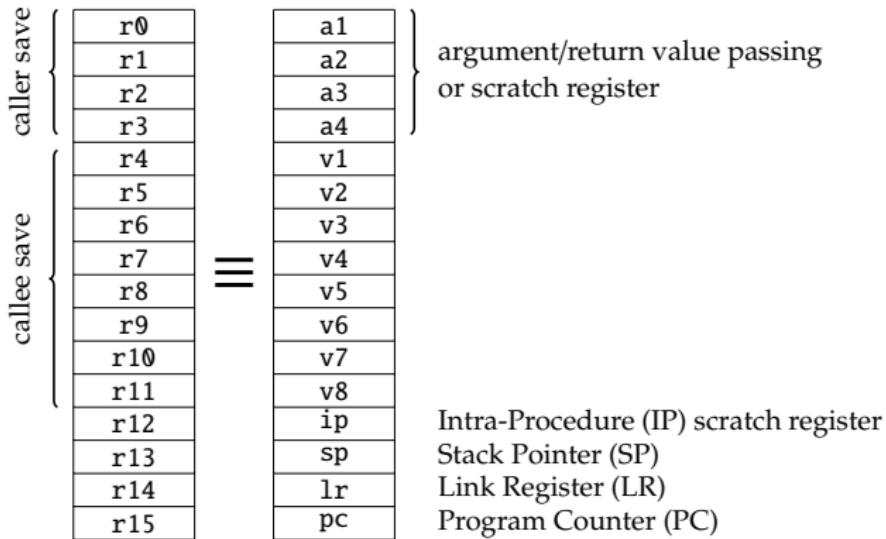
1. it's a **register machine**, and
2. it's a **load-store architecture**.

- ▶ ... we also need a tool-chain to program it:
  - ▶ although there are various ARM-specific tool-chains, e.g.,
    - ▶ ARM Developer Suite (ADS),
    - ▶ ARM Development Studio (DS), or
    - ▶ Kali MDK-ARM,
  - ▶ we'll use an open source, GCC-based alternative, **but**
  - ▶ this demands gas-style assembly language

so **beware** if you see an example somewhere else!

## ARMv7-A (2) – Registers

- ▶ ARMv7-A specifies [6, Section A2.3] a 16-entry general(ish)-purpose register file



noting

- ▶ some clearly *do* have special-purpose roles, *but*
- ▶ are generally-*addressable*, meaning the instruction set is (more or less) orthogonal.

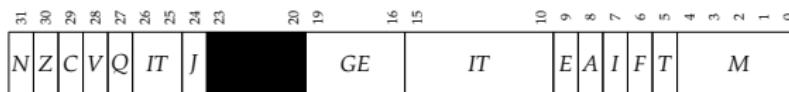
## ARMv7-A (3) – Registers

- ▶ ARMv7-A specifies [6, Section B1.3.3] two special-purpose registers
  1. a Current Program Status Register (CPSR), plus
  2. a Saved Program Status Register (SPSR)

with the latter only accessible in privileged modes.

- ▶ Note that

- ▶ the format of both CPSR and SPSR is



- ▶ transfer instructions can move a special-purpose register to

```
mrs r0, cpsr
```

and from

```
msr cpsr, r0
```

general-purpose registers, and

- ▶ writing to CPSR in user mode is limited: one cannot alter the processor mode, for example!

- ▶ Standard data processing (e.g., ALU-like) [6, Section A4.4] are available with obvious semantics, e.g.,

add r0, r1, #1	↔	GPR[0] ← GPR[1] + 1 <sub>(10)</sub>
add r0, r1, r2	↔	GPR[0] ← GPR[1] + GPR[2]
adc r0, r1, r2	↔	GPR[0] ← GPR[1] + GPR[2] + CPSR[C]
and r0, r1, r2	↔	GPR[0] ← GPR[1] ∧ GPR[2]
eor r0, r1, r2	↔	GPR[0] ← GPR[1] ⊕ GPR[2]
orr r0, r1, r2	↔	GPR[0] ← GPR[1] ∨ GPR[2]

noting that

- ▶ by default, most such instructions *don't* update flags in CPSR, *but*
- ▶ updates are enabled by an 's' suffix, e.g.,

$$\text{adds } r0, r1, r2 \mapsto \begin{cases} \text{GPR}[0] \leftarrow \text{GPR}[1] + \text{GPR}[2] \\ \text{CPSR} \leftarrow f(\text{CPSR}, \text{GPR}[1] + \text{GPR}[2]) \end{cases}$$

- ▶ The “flexible second operand” [6, Section A4.4.1] can take four forms, namely

1. an unshifted immediate value, e.g.,

$$\begin{array}{lll} \text{add } r0, r1, \#1 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + 1_{(10)} \\ \text{add } r0, r1, \#0xF & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + F_{(16)} \end{array}$$

2. an unshifted register value, e.g.,

$$\text{add } r0, r1, r2 \mapsto \text{GPR}[0] \leftarrow \text{GPR}[1] + \text{GPR}[2]$$

3. a register value shifted by an immediate value, e.g.,

$$\begin{array}{lll} \text{add } r0, r1, r2, lsl \#1 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + (\text{GPR}[2] \ll 1_{(10)}) \\ \text{add } r0, r1, r2, lsr \#1 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + (\text{GPR}[2] \gg 1_{(10)}) \\ \text{add } r0, r1, r2, ror \#1 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + (\text{GPR}[2] \ggg 1_{(10)}) \end{array}$$

4. a register value shifted by a register value, e.g.,

$$\begin{array}{lll} \text{add } r0, r1, r2, lsl r3 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + (\text{GPR}[2] \ll \text{GPR}[3]) \\ \text{add } r0, r1, r2, lsr r3 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + (\text{GPR}[2] \gg \text{GPR}[3]) \\ \text{add } r0, r1, r2, ror r3 & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] + (\text{GPR}[2] \ggg \text{GPR}[3]) \end{array}$$

- ▶ A small set of comparisons is available, i.e.,

$$\begin{array}{ll} \text{cmp } r0, r1 & \mapsto \text{CPSR} \leftarrow f(\text{CPSR}, \text{GPR}[0] - \text{GPR}[1]) \\ \text{cmn } r0, r1 & \mapsto \text{CPSR} \leftarrow f(\text{CPSR}, \text{GPR}[0] + \text{GPR}[1]) \\ \text{tst } r0, r1 & \mapsto \text{CPSR} \leftarrow f(\text{CPSR}, \text{GPR}[0] \wedge \text{GPR}[1]) \\ \text{teq } r0, r1 & \mapsto \text{CPSR} \leftarrow f(\text{CPSR}, \text{GPR}[0] \oplus \text{GPR}[1]) \end{array}$$

which

- ▶ *only* update flags in CPSR (e.g., no result is produced in a general-purpose register), and
- ▶ all have an implicit update suffix (i.e., `cmps` or similar is not required).

- ▶ Standard data movement instructions are available with obvious semantics, e.g.,
  1. immediate-to-register and register-to-register moves, e.g.,

```
mov r0, #1    ↪ GPR[0] ← 1(10)
mov r0, r1    ↪ GPR[0] ← GPR[1]
mvn r0, r1    ↪ GPR[0] ← ¬GPR[1]
```

and

2. single-shot memory accesses [6, Section A4.6], e.g.,

```
ldr r0, [ r1 ]  ↪ GPR[0] ← MEM[GPR[1]]4
str r0, [ r1 ]  ↪ MEM[GPR[1]]4 ← GPR[0]
```

*plus ...*

## ARMv7-A (8) – Data movement instructions

- ... a suite of

1. multi-shot memory accesses [6, Section A4.7], e.g.,

$$\text{ldm r0, } \{ \text{r1, r2, r3} \} \mapsto \begin{cases} \text{GPR[1]} \leftarrow \text{MEM[GPR[0] + } 0_{(10)} \text{]}^4 \\ \text{GPR[2]} \leftarrow \text{MEM[GPR[0] + } 4_{(10)} \text{]}^4 \\ \text{GPR[3]} \leftarrow \text{MEM[GPR[0] + } 8_{(10)} \text{]}^4 \end{cases}$$

$$\text{stm r0, } \{ \text{r3, r2, r1} \} \mapsto \begin{cases} \text{MEM[GPR[0] + } 0_{(10)} \text{]}^4 \leftarrow \text{GPR[1]} \\ \text{MEM[GPR[0] + } 4_{(10)} \text{]}^4 \leftarrow \text{GPR[2]} \\ \text{MEM[GPR[0] + } 8_{(10)} \text{]}^4 \leftarrow \text{GPR[3]} \end{cases}$$

and

2. multi-shot stack memory accesses, e.g.,

$$\text{push } \{ \text{r1, r2} \} \mapsto \begin{cases} t \leftarrow \text{SP} - (2 \cdot 4_{(10)}) \\ \text{MEM}[t + 0_{(10)}]^4 \leftarrow \text{GPR[1]} \\ \text{MEM}[t + 4_{(10)}]^4 \leftarrow \text{GPR[2]} \\ \text{SP} \leftarrow \text{SP} - (2 \cdot 4_{(10)}) \end{cases}$$

$$\text{pop } \{ \text{r1, r2} \} \mapsto \begin{cases} t \leftarrow \text{SP} \\ \text{MEM}[t + 0_{(10)}]^4 \leftarrow \text{GPR[1]} \\ \text{MEM}[t + 4_{(10)}]^4 \leftarrow \text{GPR[2]} \\ \text{SP} \leftarrow \text{SP} + (2 \cdot 4_{(10)}) \end{cases}$$

in a *range* of addressing modes [6, Chapter A8.55].

- Both branch and branch-and-link instructions [6, Section A4.3] are available, i.e.,

$$\begin{array}{ll} b \text{ } \text{label} & \mapsto \text{PC} \leftarrow \text{PC} + \delta(\text{PC}, \&\text{label}) \\ b \text{ } r0 & \mapsto \text{PC} \leftarrow \text{GPR}[0] \end{array}$$

$$bl \text{ } \text{function} \mapsto \begin{cases} \text{LR} \leftarrow \text{PC} + 4 \\ \text{PC} \leftarrow \text{PC} + \delta(\text{PC}, \&\text{function}) \end{cases}$$

$$bl \text{ } r0 \mapsto \begin{cases} \text{LR} \leftarrow \text{PC} + 4 \\ \text{PC} \leftarrow \text{GPR}[0] \end{cases}$$

noting that

1. label-based (resp. register-based) branches are relative (resp. absolute), and
2. a function return is simple: just write to PC directly via

```
mov pc, lr
```

or use a dedicated instruction (since ARMv4), namely

```
bx lr
```

but there are *no* conditional branches ...

- ▶ ... eh?!

- ▶ every instruction is conditionally executed, using **predicated execution** [6, Section A8.3],
- ▶ every instruction  $I$  has a 4-bit code identifying a predicate  $p$ ; you can model execution via

$$\text{if } p = \text{true} \text{ then } I \text{ else nop}$$

- ▶ the predicates are based on flags in CPSR, and specified as a suffix, e.g.,

<code>addcs r0, r1, r2</code>	$\mapsto$	<code>if CPSR[C] = 1 then add r0, r1, r2 else nop</code>
<code>bne label</code>	$\mapsto$	<code>if CPSR[Z] = 0 then b label else nop</code>

# ARMv7-A (13) – Control-flow instructions

Predicated execution

## ARMv7-A instruction predicates [6, Table A8-1]

Code	Mnemonic	Description	Predicate
0000 <sub>(2)</sub>	eq	equal	CPSR[Z] = 1
0001 <sub>(2)</sub>	ne	not equal	CPSR[Z] = 0
0010 <sub>(2)</sub>	cs (or hs)	carry set (or unsigned higher or same)	CPSR[C] = 1
0011 <sub>(2)</sub>	cc (or lo)	carry clear (or unsigned lower )	CPSR[C] = 0
0100 <sub>(2)</sub>	mi	negative	CPSR[N] = 1
0101 <sub>(2)</sub>	pl	positive	CPSR[N] = 0
0110 <sub>(2)</sub>	vs	overflow	CPSR[V] = 1
0111 <sub>(2)</sub>	vc	no overflow	CPSR[V] = 0
1000 <sub>(2)</sub>	hi	unsigned higher	CPSR[C] = 1 $\wedge$ (CPSR[Z] = 0)
1001 <sub>(2)</sub>	ls	unsigned lower	CPSR[C] = 0 $\vee$ (CPSR[Z] = 1)
1010 <sub>(2)</sub>	ge	signed greater-than or equal	CPSR[N] = CPSR[V]
1011 <sub>(2)</sub>	lt	signed less-than	CPSR[N] $\neq$ CPSR[V]
1100 <sub>(2)</sub>	gt	signed greater-than	CPSR[Z] = 0 $\wedge$ (CPSR[N] = CPSR[V])
1101 <sub>(2)</sub>	le	signed less-than or equal	CPSR[Z] = 1 $\vee$ (CPSR[N] $\neq$ CPSR[V])
1110 <sub>(2)</sub>	al	always	true
1111 <sub>(2)</sub>	nv	never	false

## Listing

```
1 int gcd( int a, int b ) {  
2     while( a != b ) {  
3         if( a > b ) {  
4             a -= b;  
5         }  
6         else {  
7             b -= a;  
8         }  
9     }  
10    return a;  
11 }
```

## Listing

```
1 loop: cmp r0, r1      ; eq if a == b  
2                      ; lt if a < b  
3                      ;  
4     beq done          ; if eq, goto done  
5                      ;  
6     blt skip           ; if lt, goto skip  
7     sub r0, r0, r1    ; true branch: a = a - b  
8     b loop            ;                      goto loop  
9 skip: sub r1, r1, r0 ; false branch: b = b - a  
10    b loop            ;                      goto loop  
11                      ;  
12 done:              ;
```

- ▶ Note that:
  - ▶ for short sequences, we avoid explicit branches (making the pipeline more effective), *but*
  - ▶ depending on the pipeline there is a  $n > 0$  cycle penalty for fetching then discarding an instruction, *plus*
  - ▶ quite often the long sequence will need to update and/or test CPSR, but this may prevent correct predication.

# ARMv7-A (14) – Control-flow instructions

Predicated execution

## Listing

```
1 int gcd( int a, int b ) {  
2     while( a != b ) {  
3         if( a > b ) {  
4             a -= b;  
5         }  
6         else {  
7             b -= a;  
8         }  
9     }  
10    return a;  
11 }
```

## Listing

```
1 loop: cmp    r0, r1      ; ne if a != b  
2                      ; gt if a > b  
3                      ; lt if a < b  
4                      ;  
5          subgt r0, r0, r1 ; if gt, true branch: a = a - b  
6          sublt r1, r1, r0 ; if lt, false branch: b = b - a  
7          bne   loop       ; if ne,  
                           goto loop
```

- ▶ Note that:
  - ▶ for short sequences, we avoid explicit branches (making the pipeline more effective), *but*
  - ▶ depending on the pipeline there is a  $n > 0$  cycle penalty for fetching then discarding an instruction, *plus*
  - ▶ quite often the long sequence will need to update and/or test CPSR, but this may prevent correct predication.

- ▶ gcc uses the `-mabi` to select between

- ▶ `apcs-gnu`,
- ▶ `atpchs`,
- ▶ `aapcs`,
- ▶ `aapcs-linux`, and
- ▶ `iwmmxt`

function calling conventions ...

- ▶ ... to be AAPCS [8] compliant, we must

1. ensure the implementations of each public interface (i.e., function) conform to the standard,
2. maintain various stack limits and alignment [8, Section 5.2.1.1],
3. observe rules about use of the ip register [8, Section 5.3.1.1], and
4. use standard rules for data types and their layout (e.g., function arguments)

plus it's useful to gdb-friendly re. back-tracing.

# ARMv7-A (16) – Control-flow instructions

## Function calls

### Listing

```
1 int callee( int a,
2             int b,
3             int c,
4             int d,
5             int e ) {
6
7     int x, y, z;
8     ...
9     return ...;
10 }
11
12 void caller() {
13     ...
14     int r = callee( ... );
15     ...
16 }
```

### Listing

```
1 callee: mov    ip, sp          ; save stack pointer
2         stmfd sp!, {v1-v7, fp, ip, lr, pc} ; save    callee-save GPRs
3
4         sub    fp, ip, #4           ; create   stack frame
5         sub    sp, sp, #12        ; set      frame pointer
6
7         ldr    v1, [ fp, #4 ]       ; create   local variable space
8         ...
9
10        ldmea fp, {v1-v7, fp, sp, pc} ; restore callee-save GPRs
11
12        ; destroy stack frame
13
14 caller: ...
15         push   {a1-a4}          ; save    caller-save GPRs
16         mov    a1, ...
17         mov    a2, ...
18         mov    a3, ...
19         mov    a4, ...
20         str    ..., [ sp, #-4 ]! ; set      argument #1
21         bl    callee            ; push    argument #2
22         mov    ..., a1           ; call
23         add    sp, sp, #4        ; save return value
24         pop    {a1-a4}          ; discard argument #3
25         ...
26
```

# ARMv7-A (16) – Control-flow instructions

## Function calls

### Listing

```
1 int gcd( int a, int b ) {
2     while( a != b ) {
3         if( a > b ) {
4             a -= b;
5         }
6         else {
7             b -= a;
8         }
9     }
10
11    return a;
12 }
13
14 void foo() {
15     ...
16     int r = gcd( 10, 20 );
17     ...
18 }
```

### Listing

```
1 gcd:   cmp    a1, a2      ; ne if a != b
2                   ; gt if a > b
3                   ; lt if a < b
4
5         ;
6         subgt a1, a1, a2 ; if gt, true branch: a = a - b
7         sublt a2, a2, a1 ; if lt, false branch: b = b - a
8         bne   gcd       ; if ne,           goto gcd
9
10        ;
11        mov    pc, lr    ; return
12
13 foo:   ...
14         mov    a1, #10    ; set a1 = a = 10
15         mov    a2, #20    ; set a2 = b = 20
16         bl    gcd       ; call
17         ...            ; use a1 = r = gcd( 10, 20 )
```

# Conclusions

- ▶ Take away points:

1. Bad news:

- ▶ elements of the unit require some low-level (e.g., assembly language) programming,
- ▶ this fact probably won't delight everyone!

2. Good news:

- ▶ the requirement above is as limited as far as is possible,
- ▶ ARM has a fairly friendly ISE, so it isn't as impenetrable as it might seem,
- ▶ there are plenty of resources available to help iff. you look,
- ▶ the skills you acquire are transferable.

## Additional Reading

- ▶ Wikipedia: ARM. . URL: [http://en.wikipedia.org/wiki/ARM\\_architecture](http://en.wikipedia.org/wiki/ARM_architecture).
- ▶ S.P. Dandamudi. “Chapter 8: ARM Architecture”. In: *Guide to RISC Processors for Programmers and Engineers*. Springer, 2004.
- ▶ A. N. Sloss, D. Symes, and C. Wright. “Chapter 2: ARM processor fundamentals”. In: *ARM System Developer’s Guide: Designing and Optimizing System Software*. Elsevier, 2004.
- ▶ A. N. Sloss, D. Symes, and C. Wright. “Chapter 3: Introduction to the ARM instruction set”. In: *ARM System Developer’s Guide: Designing and Optimizing System Software*. Elsevier, 2004.
- ▶ A. N. Sloss, D. Symes, and C. Wright. “Chapter 6: Writing and optimizing ARM assembly code”. In: *ARM System Developer’s Guide: Designing and Optimizing System Software*. Elsevier, 2004.

## References

- [1] *Wikipedia: ARM*. URL: [http://en.wikipedia.org/wiki/ARM\\_architecture](http://en.wikipedia.org/wiki/ARM_architecture) (see p. 24).
- [2] S.P. Dandamudi. “Chapter 8: ARM Architecture”. In: *Guide to RISC Processors for Programmers and Engineers*. Springer, 2004 (see p. 24).
- [3] A. N. Sloss, D. Symes, and C. Wright. “Chapter 2: ARM processor fundamentals”. In: *ARM System Developer’s Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see p. 24).
- [4] A. N. Sloss, D. Symes, and C. Wright. “Chapter 3: Introduction to the ARM instruction set”. In: *ARM System Developer’s Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see p. 24).
- [5] A. N. Sloss, D. Symes, and C. Wright. “Chapter 6: Writing and optimizing ARM assembly code”. In: *ARM System Developer’s Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see p. 24).
- [6] *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. ARM Ltd., 2014. URL: <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html> (see pp. 4–6, 8–11, 13–17).
- [7] *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. ARM Ltd., 2010. URL: <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html> (see pp. 4–6).
- [8] *Procedure Call Standard for the ARM Architecture*. Tech. rep. IHI-0042E. ARM Ltd., 2012. URL: <http://infocenter.arm.com/help/topic/com.arm.doc.ihf0042e/index.html> (see p. 20).