

Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
(csdsp@bristol.ac.uk)

February 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
 - ▶ the associated notes page may be pre-populated with extra, written explanation of material covered in lecture(s), plus
 - ▶ anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

Notes:

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

Notes:

- The goal here comes with quite a strong caveat: this can only ever be an overview, since the topic is a) far too large and b) ideally explored in more depth via hands-on, practical methods. Some topics explicitly out of scope include
 1. the so-called Unified Assembler Language (UAL) used extensively by the ARM documentation [12, Section A4.2] to specify instruction syntax and semantics,
 2. any features or extensions [12, Section A1.4] which are optional wrt. their implementation: examples include support for floating-point, Thumb, Jazelle, NEON and TrustZone.

However, the ubiquity of ARM means a *very* rich set of resources are available to supplement this overview. For example:

- Various excellent tutorials for ARM assembly language programming are available online, e.g.,

<http://www.davespace.co.uk/arm/>

- ARM offer a “quick reference” card for the ISA [10], which can be very useful when reading or writing assembly language programs:

http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf

- Saying we *need* to select a reference hardware platform might be a bit strong: we *could* obviously frame everything wrt. an abstract or even imaginary alternative, but reason doing so is harder and less compelling.

COMS20001 lecture: week #13

- ▶ 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



motivated, for example, by

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

Notes:

- The goal here comes with quite a strong caveat: this can only ever be an overview, since the topic is a) far too large and b) ideally explored in more depth via hands-on, practical methods. Some topics explicitly out of scope include
 1. the so-called Unified Assembler Language (UAL) used extensively by the ARM documentation [12, Section A4.2] to specify instruction syntax and semantics,
 2. any features or extensions [12, Section A1.4] which are optional wrt. their implementation: examples include support for floating-point, Thumb, Jazelle, NEON and TrustZone.

However, the ubiquity of ARM means a *very* rich set of resources are available to supplement this overview. For example:

- Various excellent tutorials for ARM assembly language programming are available online, e.g.,

<http://www.davespace.co.uk/arm/>

- ARM offer a “quick reference” card for the ISA [10], which can be very useful when reading or writing assembly language programs:

http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf

- Saying we *need* to select a reference hardware platform might be a bit strong: we *could* obviously frame everything wrt. an abstract or even imaginary alternative, but reason doing so is harder and less compelling.

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

Notes:

- The goal here comes with quite a strong caveat: this can only ever be an overview, since the topic is a) far too large and b) ideally explored in more depth via hands-on, practical methods. Some topics explicitly out of scope include
 1. the so-called Unified Assembler Language (UAL) used extensively by the ARM documentation [12, Section A4.2] to specify instruction syntax and semantics,
 2. any features or extensions [12, Section A1.4] which are optional wrt. their implementation: examples include support for floating-point, Thumb, Jazelle, NEON and TrustZone.

However, the ubiquity of ARM means a *very* rich set of resources are available to supplement this overview. For example:

- Various excellent tutorials for ARM assembly language programming are available online, e.g.,

<http://www.davespace.co.uk/arm/>

- ARM offer a “quick reference” card for the ISA [10], which can be very useful when reading or writing assembly language programs:

http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/QRC0001_UAL.pdf

- Saying we *need* to select a reference hardware platform might be a bit strong: we *could* obviously frame everything wrt. an abstract or even imaginary alternative, but reason doing so is harder and less compelling.

► **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**: $\text{ARM}_x \Rightarrow \text{ARM architecture version } x \approx \text{ISA version } x$, or
2. a **processor**: ARM_x ($x \in \{1, 2, \dots, 11\}$), Cortex-A/R/Mx and SCx00.

Notes:

- The term ARM architecture is intended to capture the idea that the design includes both an instruction set (an ISA) plus a programmer model (e.g., including details of the memory organisation). Since this can be confusing when set in the wider context of computer architecture, we stick with more traditional terminology and refer to both ISA (i.e., the interface between processor and software) *and* (micro-)architecture (i.e., the concrete processor implementation). Modern examples decompose the architecture into different profiles, application (A), mobile (M) or real-time (R), st. one architecture really captures a family of more specific cases. With this in mind, remember that a given ISA might be realised by many physically different micro-architectural designs: the ISA is an interface between programmer and hardware, but the hardware is free to realise the required behaviour any way it wants (e.g., to make a design choice based on some trade-off). For instance, the ARM7TDMI and ARM920T processors *both* implement the ARMv4T ISA, so are compatible wrt. the programming model. However, they make fundamentally different design choices: the former adopts a von Neumann memory organisation and 3-stage pipeline, the latter a Harvard memory organisation and 5-stage pipeline.
- In some contexts you may see the term “26-bit ARM” or similar: this is a reference to (very) old versions of the ISA (specifically ARMv2) where only a 26-bit PC was available (the remaining bits were reserved for a limited version of the CPSR register). This is basically a red herring: the design is different enough to new versions (ARMv3 and later) to confuse matters, so we ignore it entirely.
- The naming conventions ARM uses for ISAs and processors has, over time, become *very* complicated! There is an overview here

<http://community.arm.com/groups/processors/blog/2011/11/02/arm-fundamentals-introduction-to-understanding-arm-processors>

or in [5], plus Wikipedia maintains an organised list of processor and ISA versions at

http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures

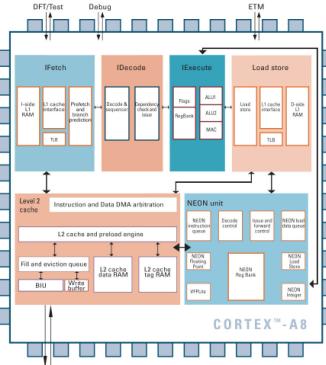
with concrete examples: for example

- a numeric prefix xyz st.
 - x is the processor family, e.g., $x = 1$ means ARM1,
 - $x1$, i.e., $y = 1$ means cache + MMU,
 - $x2$, i.e., $y = 2$ means cache + MMU + process ID,
 - $x3$, i.e., $y = 3$ means physically mapped cache + MMU,
 - $x4$, i.e., $y = 4$ means cache + MPU,
 - $x5$, i.e., $y = 5$ means physically mapped cache + MPU + ECC memory,
 - $x6$, i.e., $y = 6$ means write buffer only (no cache),
 - $x7$, i.e., $y = 7$ means physically mapped cache + MMU + AXI bus,
 - $xy6$, i.e., $z = 6$ means TCM,

and

- a (set of) suffix(es), e.g.,

- We'll focus on the 32-bit RISC(ish) Cortex-A8 [13] processor



which implements the ARMv7-A [12] 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

© Daniel Page (<https://danielpage.net/>)
Concurrent Computing (Operating Systems)

git # 1d9e4fa @ 2018-02-08

University of
BRISTOL

Notes:

- The term ARM architecture is intended to capture the idea that the design includes both an instruction set (an ISA) plus a programmer model (e.g., including details of the memory organisation). Since this can be confusing when set in the wider context of computer architecture, we stick with more traditional terminology and refer to both ISA (i.e., the interface between processor and software) *and* (micro-)architecture (i.e., the concrete processor implementation). Modern examples decompose the architecture into different profiles, application (A), mobile (M) or real-time (R), st. one architecture really captures a family of more specific cases. With this in mind, remember that a given ISA might be realised by many physically different micro-architectural designs: the ISA is an interface between programmer and hardware, but the hardware is free to realise the required behaviour any way it wants (e.g., to make a design choice based on some trade-off). For instance, the ARM7TDMI and ARM920T processors *both* implement the ARMv4T ISA, so are compatible wrt. the programming model. However, they make fundamentally different design choices: the former adopts a von Neumann memory organisation and 3-stage pipeline, the latter a Harvard memory organisation and 5-stage pipeline.
- In some contexts you may see the term "26-bit ARM" or similar: this is a reference to (very) old versions of the ISA (specifically ARMv2) where only a 26-bit PC was available (the remaining bits were reserved for a limited version of the CPSR register). This is basically a red herring: the design is different enough to new versions (ARMv3 and later) to confuse matters, so we ignore it entirely.
- The naming conventions ARM uses for ISAs and processors has, over time, become *very* complicated! There is an overview here

<http://community.arm.com/groups/processors/blog/2011/11/02/arm-fundamentals-introduction-to-understanding-arm-processors>

or in [5], plus Wikipedia maintains an organised list of processor and ISA versions at

http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures

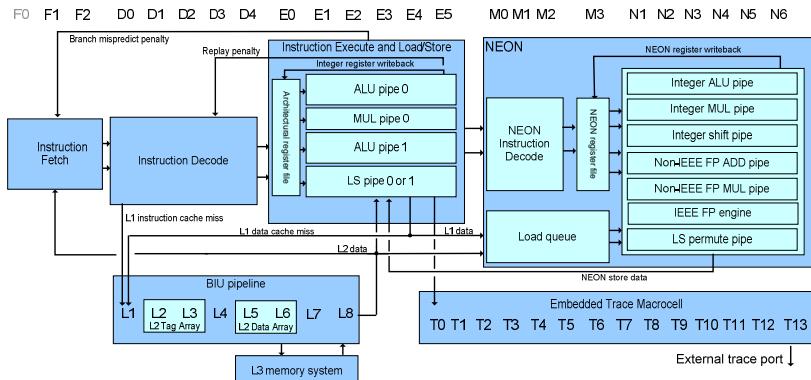
with concrete examples: for example

- a numeric prefix xyz st.
 - x is the processor family, e.g., $x = 1$ means ARM1,
 - $x1$; i.e., $y = 1$ means cache + MMU,
 - $x2$; i.e., $y = 2$ means cache + MMU + process ID,
 - $x3$; i.e., $y = 3$ means physically mapped cache + MMU,
 - $x4$; i.e., $y = 4$ means cache + MPU,
 - $x5$; i.e., $y = 5$ means physically mapped cache + MPU + ECC memory,
 - $x6$; i.e., $y = 6$ means write buffer only (no cache),
 - $x7$; i.e., $y = 7$ means physically mapped cache + MMU + AXI bus,
 - $xy6$, i.e., $z = 6$ means TCM,

and

- a (set of) suffix(es), e.g.,
 - ...

- We'll focus on the 32-bit RISC(ish) Cortex-A8 [13] processor



which implements the ARMv7-A [12] 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

© Daniel Page (<https://danielpage.net/>)
Concurrent Computing (Operating Systems)

git # 1d9e4fa @ 2018-02-08

Notes:

- The term ARM architecture is intended to capture the idea that the design includes both an instruction set (an ISA) plus a programmer model (e.g., including details of the memory organisation). Since this can be confusing when set in the wider context of computer architecture, we stick with more traditional terminology and refer to both ISA (i.e., the interface between processor and software) *and* (micro-)architecture (i.e., the concrete processor implementation). Modern examples decompose the architecture into different profiles, application (A), mobile (M) or real-time (R), st. one architecture really captures a family of more specific cases. With this in mind, remember that a given ISA might be realised by many physically different micro-architectural designs: the ISA is an interface between programmer and hardware, but the hardware is free to realise the required behaviour any way it wants (e.g., to make a design choice based on some trade-off). For instance, the ARM7TDMI and ARM920T processors *both* implement the ARMv4T ISA, so are compatible wrt. the programming model. However, they make fundamentally different design choices: the former adopts a von Neumann memory organisation and 3-stage pipeline, the latter a Harvard memory organisation and 5-stage pipeline.
- In some contexts you may see the term "26-bit ARM" or similar: this is a reference to (very) old versions of the ISA (specifically ARMv2) where only a 26-bit PC was available (the remaining bits were reserved for a limited version of the CPSR register). This is basically a red herring: the design is different enough to new versions (ARMv3 and later) to confuse matters, so we ignore it entirely.
- The naming conventions ARM uses for ISAs and processors has, over time, become *very* complicated! There is an overview here

<http://community.arm.com/groups/processors/blog/2011/11/02/arm-fundamentals-introduction-to-understanding-arm-processors>

or in [5], plus Wikipedia maintains an organised list of processor and ISA versions at

http://en.wikipedia.org/wiki/List_of_ARM_microarchitectures

with concrete examples: for example

- a numeric prefix xyz st.
 - x is the processor family, e.g., $x = 1$ means ARM1,
 - $x1$; i.e., $y = 1$ means cache + MMU,
 - $x2$; i.e., $y = 2$ means cache + MMU + process ID,
 - $x3$; i.e., $y = 3$ means physically mapped cache + MMU,
 - $x4$; i.e., $y = 4$ means cache + MPU,
 - $x5$; i.e., $y = 5$ means physically mapped cache + MPU + ECC memory,
 - $x6$; i.e., $y = 6$ means write buffer only (no cache),
 - $x7$; i.e., $y = 7$ means physically mapped cache + MMU + AXI bus,
 - $xy6$, i.e., $z = 6$ means TCM,

and

- a (set of) suffix(es), e.g.,
 - ...

- ▶ ... 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 (1) – Instruction formats

- ▶ ARMv7-A uses a fairly (!) complex instruction encoding, namely

ARMv7-A instruction formats [12, Chapter A5]

noting that

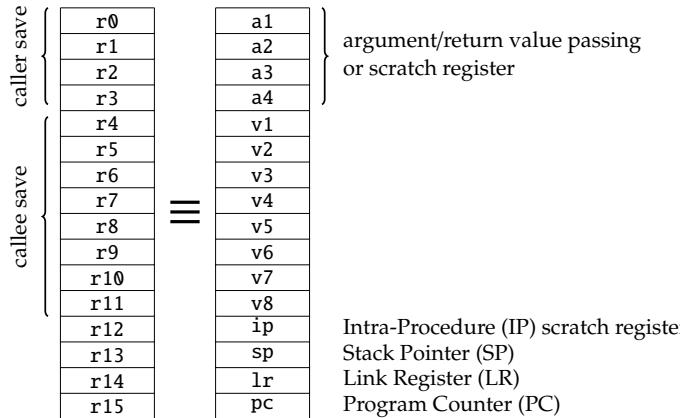
- ▶ most formats suggest 3-address style instructions, and
 - ▶ the large number of formats is (arguably) a more CISC-like characteristic.

Notes

- The fact there are multiple possible tool-chains, assemblers specifically, highlights an important point: we're *mainly* interested in behaviour of the processor, rather than how *an assembly language* might specify that behaviour (since they will differ). That's not to say we won't cover some gas syntax, but it'll be less overt than say a dedicated lecture on assembly language programming.
 - The definitive reference for gas is
<http://www.gnu.org/software/binutils/manual/>
which outlines ARM-specific details in Section 9.4.
 - The file names used for assembly language programs guide how they should be processed:
 - a file with the suffix .s should be processed by gas, whereas
 - a file with the suffix .S needs pre-processing, and would normally be processed by gcc (and then automatically by gas).

ARMv7-A (2) – Registers

- ARMv7-A specifies [12, 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.

Notes:

- Although PC is available as a general-purpose register, instructions that use r15 (or pc) as a target often have additional or special-purpose semantics; a definitive list of cases is given in [12, Section A2.3.1].
- You might see the number of general-purpose registers quoted as being larger; any larger number is likely to have also included **banked** (or **shadowed**) registers, a topic we return to when discussing interrupts.
- [14] outlines the so-called Procedure Call Standard for the ARM Architecture (AAPCS) in more detail. For example,
 - r9 (alias sb) may be used as the Static Base (SB),
 - r9 (alias tr) may be used as the Thread Register (TR),
 - r10 (alias sl) may be used as the Stack Limit (SL), and
 - r11 (alias fp) may be used as the Frame Pointer (FP)

where the use of r9 is basically platform specific; the two examples above relate to platforms that use position-independent data (cf. PIC), and thread-local storage [2]. A platform that *doesn't* define a use for r9 enables it to be used as an additional callee-save register (as illustrated). ip is an additionally interesting case: this is reserved for use by the linker [14, Section 5.3.1.1].

ARMv7-A (3) – Registers

- ARMv7-A specifies [12, Section B1.3.3] two special-purpose registers

- a Current Program Status Register (CPSR), plus
- 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!

Notes:

- You may see the term Application Program Status Register (APSR) used in certain contexts, e.g., per [12, Section A2.4]; this is often a way to distinguish between restricted access to CPSR available in user mode, and the full access in privileged mode, so it is usually reasonable to translate APSR as "CPSR in user mode" or "CPSR from the perspective of application programs".
- In detail:
 - N flag records negative result,
 - Z flag records zero result,
 - C flag records carry,
 - V flag records overflow in non-DSP operations,
 - Q flag records overflow (and saturation) in DSP operations,
 - IT controls Thumb-2 conditional execution,
 - GE controls NEON comparisons,
 - E controls the endian'ness of memory accesses to data (instruction fetches are little-endian),
 - A enables (A = 0) or disables (A = 1) imprecise aborts,
 - I enables (I = 0) or disables (I = 1) IRQ-based interrupts,
 - F enables (F = 0) or disables (F = 1) FIQ-based interrupts,
 - T || J indicates the instruction mode, st.

$$T \parallel J = \begin{cases} 00_2 & \text{ARM mode} \\ 01_2 & \text{Thumb mode} \\ 10_2 & \text{Jazelle mode} \end{cases}$$

M indicates the processor mode, st.

$$M = \begin{cases} 10000_2 & \text{user mode} \\ 10001_2 & \text{FIQ mode} \\ 10010_2 & \text{IRQ mode} \\ 10011_2 & \text{supervisor mode} \\ 10110_2 & \text{monitor mode} \\ 10111_2 & \text{abort mode} \\ 11010_2 & \text{hypervisor mode} \\ 11011_2 & \text{undefined mode} \\ 11111_2 & \text{system mode} \end{cases}$$

- Standard data processing (e.g., ALU-like) [12, Section A4.4] are available with obvious semantics, e.g.,
 - $\text{add } r0, r1, \#1 \rightarrow GPR[0] \leftarrow GPR[1] + 1_{(10)}$
 - $\text{add } r0, r1, r2 \rightarrow GPR[0] \leftarrow GPR[1] + GPR[2]$
 - $\text{adc } r0, r1, r2 \rightarrow GPR[0] \leftarrow GPR[1] + GPR[2] + CPSR[C]$
 - $\text{and } r0, r1, r2 \rightarrow GPR[0] \leftarrow GPR[1] \wedge GPR[2]$
 - $\text{eor } r0, r1, r2 \rightarrow GPR[0] \leftarrow GPR[1] \oplus GPR[2]$
 - $\text{orr } r0, r1, r2 \rightarrow GPR[0] \leftarrow GPR[1] \vee GPR[2]$

noting that

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

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

Notes:

- Use of the 's' suffix signals some special semantics when $r15$ (or pc) is used as the target. For example, whereas

$$\begin{array}{lll} \text{sub } r14, r14, \#4 & \mapsto & GPR[14] \leftarrow GPR[14] - 4_{(10)} \\ \text{subs } r14, r14, \#4 & \mapsto & \begin{cases} GPR[14] \leftarrow GPR[14] - 4_{(10)} \\ CPSR \leftarrow f(CPSR, GPR[14] - 4_{(10)}) \end{cases} \end{array}$$

we instead have that

$$\text{subs } r15, r14, \#4 \mapsto \begin{cases} GPR[15] \leftarrow GPR[14] - 4_{(10)} \\ CPSR \leftarrow SPSR \end{cases}$$

i.e., $SPSR$ (for the current mode) is copied *directly* into $CPSR$, rather than there being an *update* to flags in $CPSR$.

- Multiplication [12, Section A4.4.3] is a sort of a special case, in the sense there is

1. a "short" $32 \times 32 \rightarrow 32$ bit multiplication
 $\text{mul } r0, r1, r2 \mapsto GPR[0] \leftarrow GPR[1] \cdot GPR[2]$

which truncates the 64-bit result, i.e., forms a result from the least-significant 32-bit half, and

2. a "long" $32 \times 32 \rightarrow 64$ bit multiplication

$$\text{umull } r0, r1, r2, r3 \mapsto GPR[0] \parallel GPR[1] \leftarrow GPR[2] \cdot GPR[3]$$

which uses a 4-address instruction format to capture the 64-bit result in two registers (noting the multiplication is now unsigned).

There are also various Multiply-ACcumulate (MAC) variants, whereby a third, additional operand is added to the product computed.

ARMv7-A (5) – Data processing instructions

- The "flexible second operand" [12, Section A4.4.1] can take four forms, namely

- an unshifted immediate value, e.g.,

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

- an unshifted register value, e.g.,

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

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

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

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

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

Notes:

- In concrete terms, this is realised by placing a barrel shifter inline with the second ALU input: it acts as a sort of "pre-ALU" that can be operated in one of seven different modes, namely

lsl	=	logical left-shift
asl	=	arithmetic left-shift
lsr	=	logical right-shift
asr	=	arithmetic right-shift
ror	=	right-rotate
rrx	=	right-rotate with extend

Since there is no notion of an arithmetic left-shift, asl and lsl behave in the same way. Clearly a left-rotate by x bits can be achieved using right-rotate by $32 - x$ bits; it is not possible to right-rotate by 0 bits.

Beyond this, the right-rotate with extend mode perhaps needs further explanation. Basically, rrx and $ror \#1$ are the same *except* that the former is applied to a 33-bit value formed from the operand and $CPSR[C]$.

- Clearly there are some constraints on the shift distance:

- for shifts by an immediate value this is 5-bit, meaning a distance of $0 \leq x < 2^5 = 32$ bits, while
- for shifts by an register value this is 8-bit: the least-significant byte is used as the distance, meaning a distance of $0 \leq x < 2^8 = 256$ bits (whether or not this makes sense for larger x).

- There are various use-cases for this behaviour, including

- efficient multiplication by constants,
- for scaling (of offsets) within various addressing modes, and
- formation of constant values beyond those directly encodable as literals (due to restrictions on the bits available).

The latter is *really* important, although managed by the assembler in most cases. The idea is that the 12-bit immediate for ALU-type instructions (which includes `mov`, for example) is split so 8 bits are used as an immediate and 4 to control the barrel shifter. At face value this seems an odd choice. Even 12 bits gives $2^{12} = 4096$ values, which isn't large: 8 bits gives even less at $2^8 = 256$. However, since these 256 values can be rotated by any distance, the number of values *actually* representable is much larger.

Notes:

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

```
cmp r0, r1  ↪ CPSR ← f(CPSR, GPR[0] – GPR[1])
cmn r0, r1  ↪ CPSR ← f(CPSR, GPR[0] + GPR[1])
tst r0, r1  ↪ CPSR ← f(CPSR, GPR[0] ∧ GPR[1])
teq r0, r1  ↪ CPSR ← f(CPSR, GPR[0] ⊕ GPR[1])
```

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).

ARMv7-A (7) – Data movement instructions

Notes:

- ▶ 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 [12, 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 [12, Section A4.7], e.g.,

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

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

and

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

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

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

in a *range* of addressing modes [12, Chapter A8.55].

ARMv7-A (9) – Data movement instructions

- For a *flavour* of the address modes available, consider variants relating to

1. access size (based on a suffix)

$$\begin{array}{ll} \text{ldrdb } r0, [r1] & \mapsto GPR[0] \leftarrow \text{ext}_{32}^0(\text{MEM}[GPR[1]]^1) \\ \text{ldrsb } r0, [r1] & \mapsto GPR[0] \leftarrow \text{ext}_{32}^{\pm}(\text{MEM}[GPR[1]]^1) \\ \text{ldrh } r0, [r1] & \mapsto GPR[0] \leftarrow \text{ext}_{32}^0(\text{MEM}[GPR[1]]^2) \\ \text{ldrsh } r0, [r1] & \mapsto GPR[0] \leftarrow \text{ext}_{32}^{\pm}(\text{MEM}[GPR[1]]^2) \end{array}$$

2. indexed, scaled, pre- and post- auto-indexing access types (based on some syntax)

$$\begin{array}{ll} \text{str } r0, [r1] & \mapsto \text{MEM}[GPR[1]]^4 \leftarrow GPR[0] \\ \text{str } r0, [r1, r2] & \mapsto \text{MEM}[GPR[1] + GPR[2]]^4 \leftarrow GPR[0] \\ \text{str } r0, [r1, r2, ls1 \#1] & \mapsto \text{MEM}[GPR[1] + (GPR[2] \ll 1_{(10)})]^4 \leftarrow GPR[0] \end{array}$$

$$\text{str } r0, [r1, r2]! \mapsto \begin{cases} \text{MEM}[GPR[1] + GPR[2]]^4 \leftarrow GPR[0] \\ GPR[1] \leftarrow GPR[1] + GPR[2] \end{cases}$$

$$\text{str } r0, [r1], r2 \mapsto \begin{cases} \text{MEM}[GPR[1]]^4 \leftarrow GPR[0] \\ GPR[1] \leftarrow GPR[1] + GPR[2] \end{cases}$$

including immediate versions of each case.

Notes:

- In the multi-shot case, the register list is encoded as a bit-set x , st. if $x_i = 1$ then $GPR[i]$ is accessed, otherwise it is not. This has some implications, namely a) the order of registers names in the list makes no difference (the access order is always low-to-high), and b) using a register name more than once makes no difference (although it is always termed a register *list*, in reality it's a register *set*). One obvious restriction is enforced: the base register cannot be included in the register list for some addressing modes, e.g.,

`ldm r0!, r0, r1, r2`

would be disallowed.

- If you interpret the semantics of various `ldm`-based addressing modes, one rule-of-thumb is that lower (resp. higher) register numbers are always stored at lower (resp. higher) addresses in memory. Per the above, this is true no matter what order registers are included in the register list.
- A nice short-hand for the register list allows ranges to be specified: we have that

`ldm r0, r1-r2, r3, r4, r9 ≡ ldm r0, r1-r4, r9`

for example.

- Although there is no example here, multi-shot load and store instructions allow an analogy of the 's' suffix wrt. arithmetic. This addressing mode (e.g., see [12, Section B9.3.5]) will be triggered if a '^' suffix is added to the register list, e.g.,

`ldm r0, r1, r2, r3, r4, r9 ^`

and implies altered semantics as follows:

1. for a PC-loading `ldm` instruction it means CPSR is copied from SPSR, and
2. for a non PC-loading `ldm` instruction or a `stm` instruction, the user mode registers are accessed (rather than banked registers relating to the current, non-user mode).

Notes:

- Recall that an address x is said to be w aligned if it is a multiple of w ; byte, half-word and word alignment therefore mean $w = 1$, $w = 2$, and $w = 4$ respectively, plus we can formal say that x is aligned iff. $x \equiv 0 \pmod{w}$. Prior to ARMv6, memory access had to be aligned to an appropriate w : if you load a 16-bit half-word, e.g.,

`ldrh r0, [r1]`

then the effective address x must be a multiple of 2, otherwise, if x is *unaligned* (i.e., $x \not\equiv 0 \pmod{w}$), an exception results. ARMv6 relaxed this restriction by allowing an unaligned access.

As an aside, and ignoring Thumb mode, instruction fetches still require aligned addresses. Since all instructions use a 32-bit format, there is a requirement that $\text{PC} \bmod 4 = 0$ (i.e., $\text{LSB}_2(\text{PC}) = 0$). This prevents, for example, performing a branch to an unaligned address which is sort of "half way through" a value instruction.

- The multi-shot accesses come in four (suffixed) types, namely

1. Increment After (IA), e.g.,

$$\text{stmia } r0!, \{ r1, r2 \} \mapsto \begin{cases} t \leftarrow \text{GPR}[0] \\ \text{MEM}[t + 0_{(10)}]^4 \leftarrow \text{GPR}[1] \\ \text{MEM}[t + 4_{(10)}]^4 \leftarrow \text{GPR}[2] \\ \text{GPR}[0] \leftarrow \text{GPR}[0] + (2 \cdot 4_{(10)}) \end{cases}$$

2. Decrement After (DA), e.g.,

$$\text{stmda } r0!, \{ r1, r2 \} \mapsto \begin{cases} t \leftarrow \text{GPR}[0] - (2 \cdot 4_{(10)}) + 4_{(10)} \\ \text{MEM}[t + 0_{(10)}]^4 \leftarrow \text{GPR}[1] \\ \text{MEM}[t + 4_{(10)}]^4 \leftarrow \text{GPR}[2] \\ \text{GPR}[0] \leftarrow \text{GPR}[0] - (2 \cdot 4_{(10)}) \end{cases}$$

3. Increment Before (IB), e.g.,

$$\text{stmib } r0!, \{ r1, r2 \} \mapsto \begin{cases} t \leftarrow \text{GPR}[0] + 4_{(10)} \\ \text{MEM}[t + 0_{(10)}]^4 \leftarrow \text{GPR}[1] \\ \text{MEM}[t + 4_{(10)}]^4 \leftarrow \text{GPR}[2] \\ \text{GPR}[0] \leftarrow \text{GPR}[0] + (2 \cdot 4_{(10)}) \end{cases}$$

4. Decrement Before (DB), e.g.,

$$\text{stmdb } r0!, \{ r1, r2 \} \mapsto \begin{cases} t \leftarrow \text{GPR}[0] - (2 \cdot 4_{(10)}) \\ \text{MEM}[t + 0_{(10)}]^4 \leftarrow \text{GPR}[1] \\ \text{MEM}[t + 4_{(10)}]^4 \leftarrow \text{GPR}[2] \\ \text{GPR}[0] \leftarrow \text{GPR}[0] - (2 \cdot 4_{(10)}) \end{cases}$$

where IA-type accesses are the default, i.e.,

© Daniel Page (csdp@bristol.ac.uk)
Concurrent Computing (Operating Systems)

git # 1d9e4fa @ 2018-02-08



ARMv7-A (11) – Control-flow instructions

- Both branch and branch-and-link instructions [12, Section A4.3] are available, i.e.,

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

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

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

noting that

- label-based (resp. register-based) branches are relative (resp. absolute), and
- 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 ...

Notes:

- One can characterise a stack (then termed the **stack discipline**) as

- “full”, meaning SP points at the last used element (i.e., ToS), vs.
- “empty”, meaning SP points at the next free element,

and

- “descending”, meaning it grows downward, vs.
- “ascending”, meaning it grows upward,

There are aliases for the native multi-shot load and store instructions which model these types of stack: although a full descending stack is a usual default (e.g., per [14]), you can use

- `ldmfd` and `stmdfd` to mean `ldmdb` and `stmia`,
- `lmed` and `lmed` to mean `ldmda` and `stmib`,
- `ldmfa` and `stmfda` to mean `ldmb` and `stmda`, and
- `ldmea` and `stmea` to mean `ldmia` and `stmdb`.

Based on this description, it should be clear that in fact

$$\begin{array}{ll} \text{push } \{ r1, r2 \} & \equiv \text{stmdb sp!}, \{ r0, r1 \} \\ \text{pop } \{ r1, r2 \} & \equiv \text{ldmia sp!}, \{ r0, r1 \} \end{array}$$

since these (pseudo-)instructions assume a full descending stack.

Notes:

- A relative branch is limited wrt. the maximum distance (or offset) that can be realised. Actually computing that maximum needs quite some detail. Recall that both the branch and branch-and-link instructions are encoded using a format with a 24-bit, signed immediate. The actual distance is computed as

$$\delta(\text{PC}, \&\text{label}) - 8$$

to accommodate the effect of pipelining; during decoding, the resulting immediate is sign-extended. Since this will always yield a multiple of 32-bits (or 4 bytes) due to the need for alignment of PC, the 2 LSBs do not need to be stored and so 2 more MSBs can be. As a result, the offset is limited by

$$-2^{25} \leq x < 2^{25} - 1$$

words which is to say it can be (roughly) $\pm 32\text{MB}$. Clearly that might not *always* be enough, but certainly it gives enough range for most common use-cases.

- `gas` supports a special “full stop” (or “dot”) symbol which refers to the current address. As such,

`b .`

is a branch instruction to the current address, or, if you prefer, a branch with zero offset from the current value of PC: you could also term this an infinite loop!

ARMv7-A (12) – Control-flow instructions

Predicated execution

► ... eh?!

- every instruction is conditionally executed, using **predicated execution** [12, Section A8.3],
- every instruction I has a 4-bit code identifying a predicate p ; you can model execution via

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

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

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

Notes:

- The concept of predicated execution isn't unique to ARM processors; there is an analogue in the Intel Itanium processor [3], for example, that represents a more general approach.

ARMv7-A (13) – Control-flow instructions

Predicated execution

ARMv7-A instruction predicates [12, Table A8-1]

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

Notes:

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      ; 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   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.

Notes:

- The fact there is a penalty for fetching and discarding each instruction whose predicate fails, it *could* be the case that a long predicated sequence takes longer to execute than a single branch (which is somewhat counter-intuitive if you have assumed not executed implies zero execution time).

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

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

function calling conventions ...

- ▶ ... to be AAPCS [14] 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 [14, Section 5.2.1.1],
3. observe rules about use of the ip register [14, 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.

Notes:

- A (somewhat) subtle point is that AAPCS *only* places constraints on the public interface: internal functions cannot be linked against, so you can basically do what you want!
- In some more detail, but still less than [14] itself:
 - The stack model used is full-descending.
 - The stack pointer must (universally) be a) within the stack extent, and b) be word aligned (i.e., $sp \equiv 0 \pmod{4}$); at a public interface, it must (also) be double-word aligned (i.e., $sp \equiv 0 \pmod{8}$).
 - ip may be used by the linker between callee and caller (e.g., to cope with the limited range of immediate in the use of bl), but used as a caller-save scratch register otherwise.
 - The caller-save registers a1 through a4 are used to pass arguments (resp. return values) from (resp. to) callee (resp. caller) per [14, Sections 5.4 and 5.5]; the return type dictates which registers are used, and note that use of more than 4 arguments dictates spilling the excess only.
 - the callee must preserve v1 through v5 plus v6 (where appropriate), v7, v8 and sp st. the state on return to the caller matches that before the call.

Note that AAPCS does not enforce use of a frame pointer, but clearly it *can* be used by GCC so needs to be considered re. consistency.

ARMv7-A (16) – Control-flow instructions

Listing

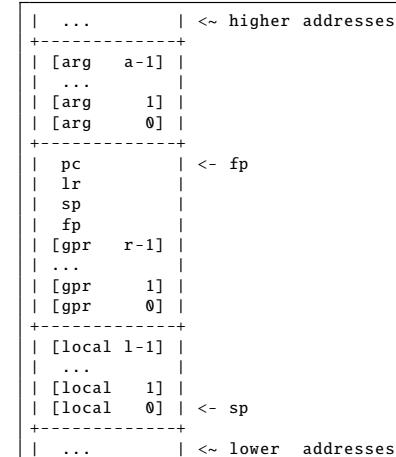
```
1 int callee( int a,
2     int b,
3     int c,
4     int d,
5     int e ) {
6
7     int x, y, z;
8 ...
9     return ...;
10 }
```

Listing

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

Notes:

- The frame layout produced by caller and callee can be illustrated as follows



noting that

- components in brackets are optional (e.g., only some arguments might be pushed into the frame since some are communicated via registers, only some general-purpose callee-save registers actually need to be saved, only space for actual local variables need be allocated), and
- any arguments into the frame can be accessed via a positive offset from fp.

- There are some reasonable questions about callee:

- Why push ip not sp? At the point ip is pushed their value is equal, so this at least works; ip is used because sp cannot be in the register list and used as the base address which is written-back to.

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;
12 }
13
14 void foo() {
15 ...
16     int r = gcd( 10, 20 );
17 ...
18 }
```

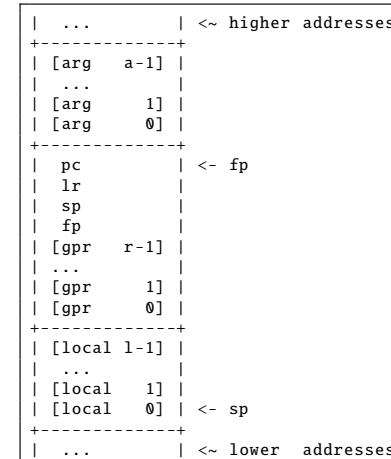
Listing

```

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

Notes:

- The frame layout produced by caller and callee can be illustrated as follows



noting that

- components in brackets are optional (e.g., only some arguments might be pushed into the frame since some are communicated via registers, only some general-purpose callee-save registers actually need to be saved, only space for actual local variables need be allocated), and
- any arguments into the frame can be accessed via a positive offset from fp.

- There are some reasonable questions about callee:

- Why push ip not sp? At the point ip is pushed their value is equal, so this at least works; ip is used because sp cannot be in the register list and used as the base address which is written-back to.

An Aside: influence of the tool-chain

- **Beware:** in various situations, you might not end up with the machine-code you expected, e.g.,

- clearly one cannot encode

```
mov r0, #0xFFFFFFFF
```

since the 32-bit literal specified cannot be accommodated, but the same result can be obtained via

```
mvn r0, #0
```

- a **pseudo-instruction** looks like an instruction, but may be translated into (or an alias for) something else ...
- ... the nop pseudo-instruction has various possible encodings, such as `mov r0, r0`, whereas examples like

```
ldr r0, =literal
ldr r0, =label
```

are dealt with by the assembler by

- trying to form a 1-instruction encoding (resp. address), likely via `mov`, otherwise
- generating a *real* `ldr` instruction from a relative address in the (managed) literal-pool.

Notes:

- ▶ Most recent versions extend the core ISA with *another*, sub-ISA called **Thumb** (and latterly **Thumb-2**):
 1. the ARM instruction set has 32-bit fixed-length formats, and
 2. the Thumb instruction set has “reduced” 16-bit fixed-length formats
- ▶ **Question:** *why?*
- ▶ **Answer:** code density and memory access cost, e.g.,
 1. Thumb (resp. Thumb-2) can improve the number of instructions per unit of memory by 30% (resp. 70%), and
 2. 16-bit instructions allow a narrower memory interface, or, conversely, fewer memory accesses per instruction

both of which are important in embedded contexts (cf. power consumption).

Notes:

- ▶ This *isn't* really variable-length encoding:
 - ▶ The processor is *either* in ARM mode *or* in Thumb mode (st. the decoder works differently) allowing a choice.
 - ▶ A change of mode is invoked by a `bx` instruction; the mode is reflected by CPSR[T] flag in CPSR.
- ▶ Internally, there is only one set of micro-instructions but the two ISAs allow their specification in different ways:
 - ▶ the easiest metaphor is the 16-bit instructions being a form of “short-hand” for the 32-bit instructions,
 - ▶ often they are 2-address rather than 3-address, and
 - ▶ a sub-set of features is available (e.g., no predicated execution, can't access all registers).

Notes:

- Other processor designs now include similar techniques; for example the MIPS16e ISA is a similar concept (as used in MIPS processors).
- Technologies like IBM CodePack [9] (as used in PowerPC processors) go further by compressing instructions at compile-time, then decompressing them at run-time.

► 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.

Notes:

Additional Reading

- Wikipedia: ARM. . URL: 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.

Notes:

References

- [1] Wikipedia: ARM. URL: http://en.wikipedia.org/wiki/ARM_architecture (see p. 63).
- [2] Wikipedia: Thread local storage. URL: http://en.wikipedia.org/wiki/Thread-local_storage (see p. 22).
- [3] S.P. Dandamudi. "Chapter 7: Itanium Architecture". In: *Guide to RISC Processors for Programmers and Engineers*. Springer, 2004 (see p. 42).
- [4] S.P. Dandamudi. "Chapter 8: ARM Architecture". In: *Guide to RISC Processors for Programmers and Engineers*. Springer, 2004 (see p. 63).
- [5] A. N. Sloss, D. Symes, and C. Wright. "Appendix C: Processors and architecture". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see pp. 12, 14, 16).
- [6] 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. 63).
- [7] 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. 63).
- [8] 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. 63).
- [9] A. Orpaz and S. Weiss. "A study of CodePack: optimizing embedded code space". In: *Hardware/Software (CODES)*. 2002, pp. 103–108 (see p. 60).
- [10] ARM Limited. *ARM and Thumb-2 Instruction Set Quick Reference Card*. Tech. rep. QRC-0001M. <http://infocenter.arm.com/help/topic/com.arm.doc.qrc0001m/index.html>. 2008 (see pp. 6, 8, 10).
- [11] ARM Limited. *ARM Architecture Reference Manual*. Tech. rep. DDI-0100I. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0100i/index.html>. 2005 (see p. 20).
- [12] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition*. Tech. rep. DDI-0406C. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html>. 2014 (see pp. 6, 8, 10, 11, 13, 15, 19–27, 31, 33, 34, 39, 41, 43).

Notes:

References

- [13] ARM Limited. *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html>. 2010 (see pp. 11, 13, 15).
- [14] ARM Limited. *Procedure Call Standard for the ARM Architecture*. Tech. rep. IHI-0042E. <http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/index.html>. 2012 (see pp. 22, 38, 49, 50).

Notes: