- 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.

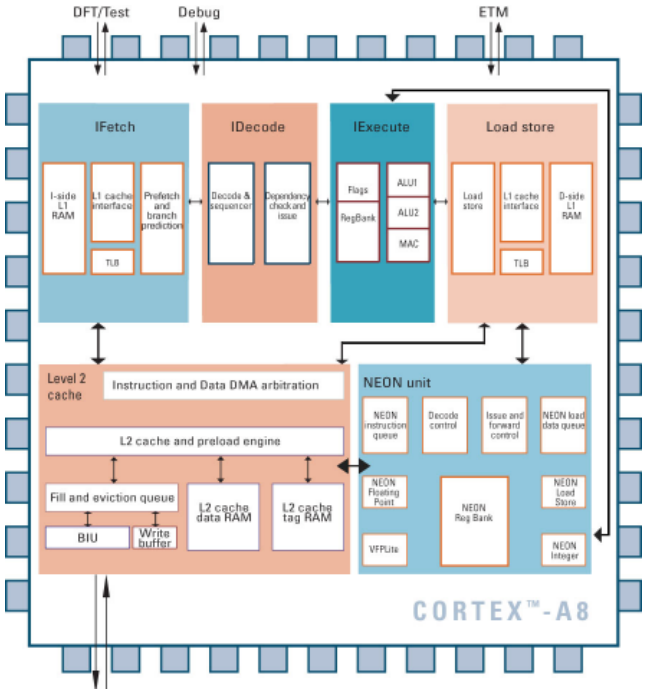
- Goal: brief, high-level overview of ARM-based
  1. processor design and capabilities, plus
  2. assembly language programming.

- In fact, saying "ARM" is imprecise: it can mean
  1. an **ISA**: ARMvx $\Rightarrow$ ARM architecture version $x \simeq$ ISA version x, *or*
  2. a **processor**: ARMx ($x \in \{1, 2, \ldots, 11\}$), Cortex-A/R/M$x$ and SCx00.

---

http://www.arm.com/files/pdf/ARM_Arch_A8.pdf

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

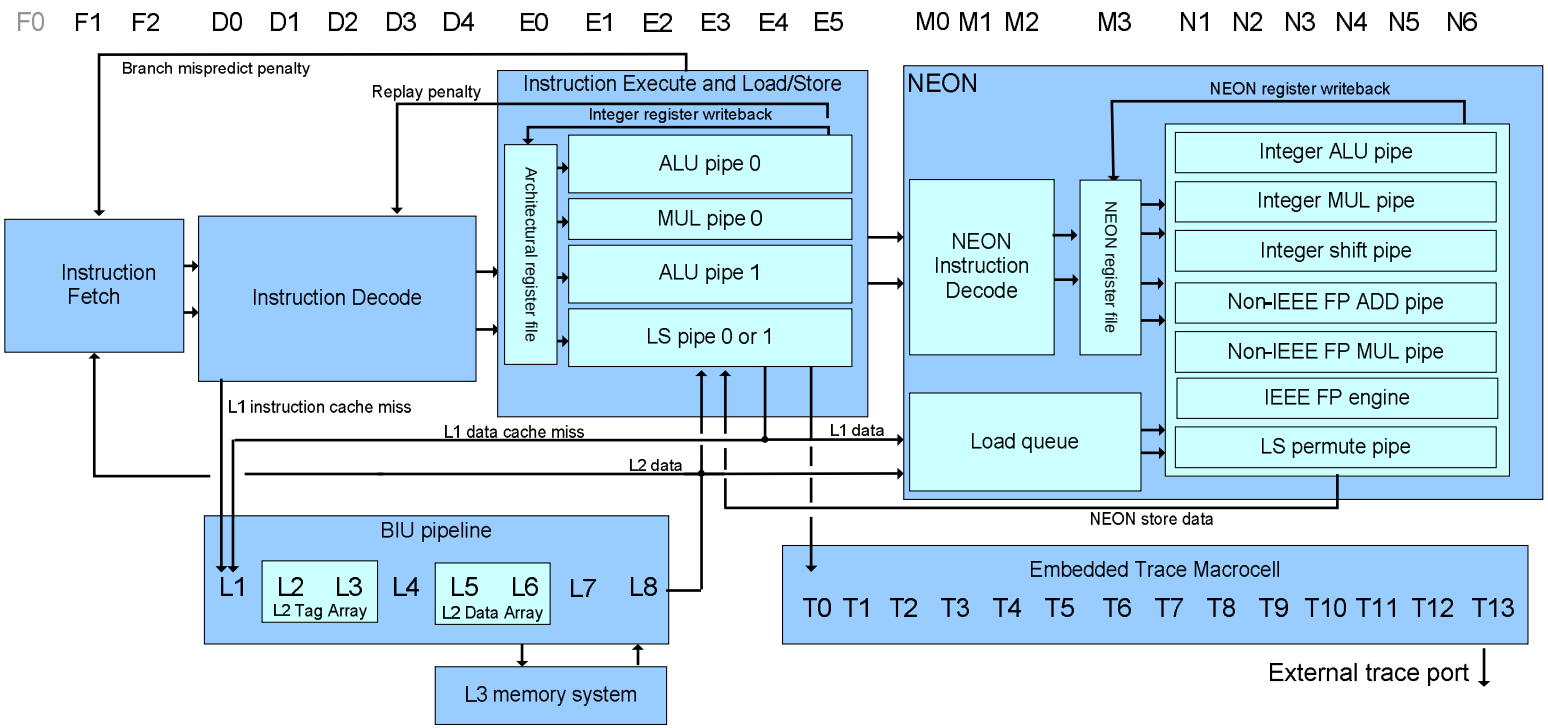


which implements the **ARMv7-A** ISA ...

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

http://www.arm.com/files/pdf/ARM_Arch_A8.pdf

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



which implements the **ARMv7-A** ISA ...

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

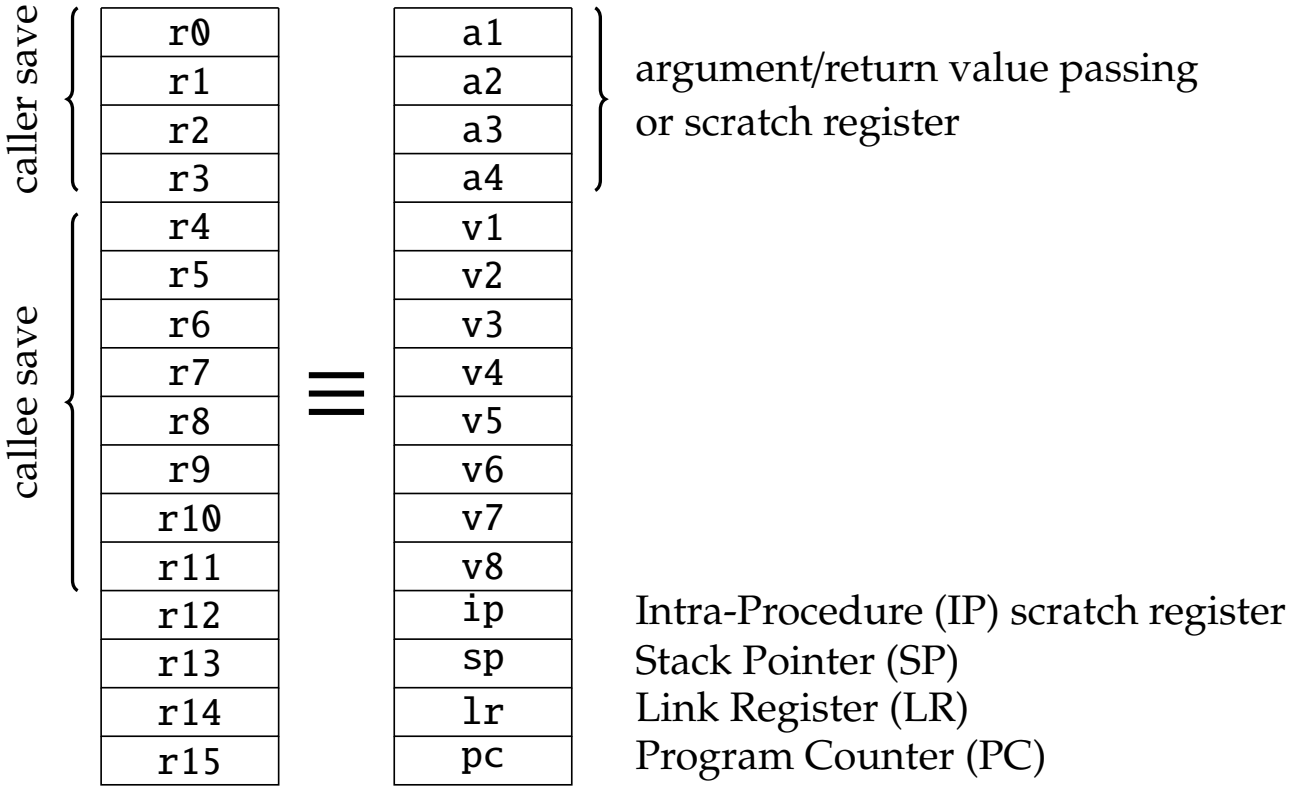
http://www.arm.com/files/pdf/ARM_Arch_A8.pdf

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

  so beware if you see an example somewhere else!

ARMv7-A (2) – Registers

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



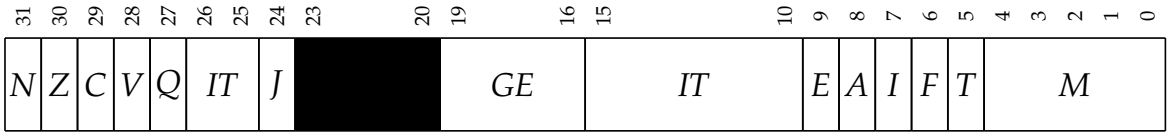
noting

- 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 [3, 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!

ARMv7-A (4) – Data processing instructions

- Standard data processing (e.g., ALU-like) [3, Section A4.4] are available with obvious semantics, e.g.,

$$
\begin{aligned}
\texttt{add r0, r1, \#1} &\mapsto \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] + 1_{(10)} \\
\texttt{add r0, r1, r2} &\mapsto \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] + \mathsf{GPR}[2] \\
\texttt{adc r0, r1, r2} &\mapsto \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] + \mathsf{GPR}[2] + \mathsf{CPSR}[C] \\
\texttt{and r0, r1, r2} &\mapsto \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] \wedge \mathsf{GPR}[2] \\
\texttt{eor r0, r1, r2} &\mapsto \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] \oplus \mathsf{GPR}[2] \\
\texttt{orr r0, r1, r2} &\mapsto \mathsf{GPR}[0] \leftarrow \mathsf{GPR}[1] \vee \mathsf{GPR}[2]
\end{aligned}
$$

  noting that

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

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

ARMv7-A (5) – Data processing instructions

- The "flexible second operand" [3, Section A4.4.1] can take four forms, namely
  1. an unshifted immediate value, e.g.,

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

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

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

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

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

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

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

ARMv7-A (6) – Data processing instructions

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

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

  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) – Control-flow instructions

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

$$\texttt{b label} \quad \mapsto \quad \mathsf{PC} \leftarrow \mathsf{PC} + \delta(\mathsf{PC}, \texttt{\&label})$$

$$\texttt{b r0} \quad \mapsto \quad \mathsf{PC} \leftarrow \mathsf{GPR}[0]$$

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

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

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

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

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

$$\texttt{addcs r0, r1, r2} \quad \mapsto \quad \textbf{if } \text{CPSR}[C] = 1 \textbf{ then } \texttt{add r0, r1, r2} \textbf{ else } \texttt{nop}$$
$$\texttt{bne label} \quad \mapsto \quad \textbf{if } \text{CPSR}[Z] = 0 \textbf{ then } \texttt{b label} \textbf{ else } \texttt{nop}$$

Listing (C)

```
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 }
```

Listing (asm)

```
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.

Listing (C)

```
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 }
```

Listing (asm)

```
1 loop: cmp   r0, r1      ; ne if a != b
2                         ; gt if a >  b
3                         ; le if a <= b
4                         ;
5       subgt r0, r0, r1  ; if gt, true  branch: a = a - b
6       suble r1, r1, r0  ; if le, 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.

ARMv7-A (11) – Data movement instructions

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

$$\begin{array}{lcl} \texttt{mov r0, \#1} & \mapsto & \text{GPR}[0] \leftarrow 1_{(10)} \\ \texttt{mov r0, r1} & \mapsto & \text{GPR}[0] \leftarrow \text{GPR}[1] \\ \texttt{mvn r0, r1} & \mapsto & \text{GPR}[0] \leftarrow \neg\text{GPR}[1] \end{array}$$

     and
  2. single-shot memory accesses [3, Section A4.6], e.g.,

$$\begin{array}{lcl} \texttt{ldr r0, [r1]} & \mapsto & \text{GPR}[0] \leftarrow \text{MEM}[\text{GPR}[0]]^4 \\ \texttt{str r0, [r1]} & \mapsto & \text{MEM}[\text{GPR}[0]]^4 \leftarrow \text{GPR}[0] \end{array}$$

  plus ...

ARMv7-A (12) – Data movement instructions

- ... a suite of
  1. multi-shot memory accesses [3, Section A4.7], e.g.,

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

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

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

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

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

  in a *range* of addressing modes [3, Chapter A8.55].

An Aside: function calling convention(s)

- `gcc` uses the `-mabi` to select between
  - `apcs-gnu`,
  - `atpcs`,
  - `aapcs`,
  - `aapcs-linux`, and
  - `iwmmxt`

  function calling conventions (!) ...
- ... to be AAPCS [2] 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 [2, Section 5.2.1.1],
  3. observe rules about use of the `ip` register [2, 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.

An Aside: function calling convention(s)

Listing (C)

```
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 (asm)

```
1  callee: mov   ip, sp                          ; save stack pointer
2          stmfd sp!, {v1-v7, fp, ip, lr, pc}    ; save    callee-save GPRs
3                                                ; create  stack frame
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          ...                                   ;      function body
9
10         ldmea fp,  {v1-v7, fp, sp, pc    }    ; restore callee-save GPRs
11                                               ; destroy stack frame
12                                               ; return
13
14 caller: ...
15         push  {a1-a4}                         ; save    caller-save GPRs
16         mov   a1, ...                         ; set     argument #1
17         mov   a2, ...                         ; set     argument #2
18         mov   a3, ...                         ; set     argument #3
19         mov   a4, ...                         ; set     argument #4
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         ...
```

An Aside: function calling convention(s)

Listing (C)

```
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 (asm)

```
1  gcd:  cmp   a1, a2      ; ne if a != b
2                          ; gt if a >  b
3                          ; le if a <= b
4                          ;
5        subgt a1, a1, a2  ; if gt, true  branch: a = a - b
6        suble a2, a2, a1  ; if le, 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 )
```

Conclusions

- Take away points:
  1. ARM offer interesting, industrial-standard ISA and processor designs ...

Conclusions

- Take away points:

2. Bad news:
   - elements of the unit require some low-level (e.g., assembly language) programming,
   - this fact probably won't delight everyone!
3. Good news:
   - this requirement is as limited as 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.

References

[1] ARM Limited.
Cortex-A8 Technical Reference Manual.
Technical Report DDI-0344K, 2010.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html.

[2] ARM Limited.
Procedure Call Standard for the ARM Architecture.
Technical Report IHI-0042E, ver. 2.09, 2012.
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042e/index.html.

[3] ARM Limited.
ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition.
Technical Report DDI-0406C, 2014.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html.

[4] S.P. Dandamudi.
Chapter 8: ARM architecture.
In *Guide to RISC Processors for Programmers and Engineers*. Springer, 2004.

[5] A. N. Sloss, D. Symes, and C. Wright.
ARM System Developer's Guide: Designing and Optimizing System Software.
Elsevier, 2004.

[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* [5].

References

[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* [5].

[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* [5].