System Construction

Autumn Semester 2015
Felix Friedrich
Goals

- Competence in building custom system software \textit{from scratch}
- Understanding of „how it really works“ behind the scenes \textit{across all levels}
- Knowledge of the approach of fully managed \textit{lean} systems

A lot of this course is \textit{about detail}.
A lot of this course is about \textit{bare metal programming}. 
Course Concept

- Discussing elaborated case studies
  - In theory (lectures)
  - and practice (hands-on lab)
- Learning by example vs. presenting topics
Prerequisite

- Knowledge corresponding to lectures *Systems Programming* and/or *Operating Systems*
  - Do you know what a stack-frame is?
  - Do you know how an interrupt works?
  - Do you know the concept of virtual memory?
- Good reference for recapitulation: *Computer Systems – A Programmer's Perspective*
Links

- SVN repository
  https://svn.inf.ethz.ch/svn/lecturers/vorlesungen/trunk/syscon/2015/shared

- Links on the course homepage
  http://lec.inf.ethz.ch/syscon/2015
Background: Co-Design @ ETH

Languages (Pascal Family)

- Modula → Oberon → ActiveOberon
- +MathOberon → Oberon07 → Zonnon
- Active Cells

Operating / Runtime Systems

- Medos → Oberon → Aos
- A2 → SoC
- HeliOs → Minos

Hardware

- Lilith → Ceres
- x86 / IA64 / ARM Emulations on Unix / Linux
- TRM (FPGA)
- RISC (FPGA)

1980 - 1990 - 2000 - 2010
Course Overview
Part 1: Contemporary Hardware

Case Study 1. Minos: Embedded System

- Safety-critical and fault-tolerant monitoring system
- Originally invented for autopilot system for helicopters
- Topics: ARM Architecture, Cross-Development, Object Files and Module Loading, Basic OS Core Tasks (IRQs, MMUs etc.), Minimal Single-Core OS: Scheduling, Device Drivers, Compilation and Runtime Support.

- New: Now with hands-on lab on Raspberry Pi (2)
Course Overview

Part 1: Contemporary Hardware

Case Study 2. A2: Multiprocessor OS

- Universal operating system for symmetric multiprocessors (SMP)
- Based on the co-design of a programming language (Active Oberon) and operating system (A2)

Case Study 2a: Lock-free Operating System Kernel

- With hands-on labs on x86ish hardware and Raspberry Pi
Case Study 3. RISC: Single-Processor System

- RISC single-processor system designed from scratch: hardware on FPGA
- Graphical workstation OS and compiler ("Project Oberon")
- Topics: building a system from scratch, Art of simplicity, Graphical OS, Processor Design.

Case Study 4. Active Cells: Multi-Processor System

- Special purpose heterogeneous system on a chip (SoC)
- Massively parallel hard- and software architecture based on Message Passing
- Topics: Dataflow-Computing, Tiny Register Machine: Processor Design Principles, Software-/Hardware Codesign, Hybrid Compilation, Hardware Synthesis
Organization

- Lecture Tuesday 13:15-15:00 (CAB G 57)
  with a break around 14:00

- Exercise Lab Tuesday 15:00 – 17:00 (CAB G 56)
  Guided, open lab, duration normally 2h
  First exercise: today (15th September)

- Oral Examination in examination period after semester (15 minutes).
  Prerequisite: knowledge from both course and lab
Design Decisions: Area of Conflict

Programming Model
Compiler
Language
Tools
System

simple / undersized

sophisticated / complex

tailored / non-generic

universal / overly generic

comprehensible / simplistic

elaborate / incomprehensible

customizable / inconvenient

feature rich / predetermined

economic / unoptimized

optimized / uneconomic

I am about here
1. CASE STUDY MINOS
Focus Topics

- Hardware platform
- Cross development
- Simple modular OS
- Runtime Support
- Realtime task scheduling
- I/O (SPI, UART)*
- Filesystem (flash disk)

*Serial Peripheral Interface, Universal Asynchronous Receiver Transmitter
Learn to Know the Target Architecture

1.1 HARDWARE
ARM Processor Architecture Family

- **32 bit Reduced Instruction Set Computer** architecture by ARM Holdings
  - 1st production 1985 (Acorn Risc Machine at 4MHz)
  - ARM Ltd. today does not sell hardware but (licenses for) chip designs
- **StrongARM**
  - by DEC & Advanced Risc Machines.
  - XScale implementation by Intel (now Marvell) after DEC take over
- More than 90 percent of the sold mobile phones (since 2007) contain at least one ARM processor (often more)*
  [95% of smart phones, 80% of digital cameras and 35% of all electronic devices*]
- Modular approach:
  ARM families produced for different profiles, such as Application Profile, Realtime Profile and Microcontroller / Low Cost Profile

*http://arm.com/about/company-profile/index.php
## ARM Architecture Versions

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM v1-3</td>
<td>Cache from ARMv2a, 32-bit ISA in 26-bit address space</td>
</tr>
<tr>
<td>ARM v4</td>
<td>Pipeline, MMU, 32 bit ISA in 32 bit address space</td>
</tr>
<tr>
<td>ARM v4T</td>
<td>16-bit encoded Thumb Instruction Set</td>
</tr>
<tr>
<td>ARM v5TE</td>
<td>Enhanced DSP instructions, in particular for audio processing</td>
</tr>
<tr>
<td>ARM v5TEJ</td>
<td>Jazelle Technology extension to support Java acceleration technology (documentation restricted)</td>
</tr>
<tr>
<td>ARM v6</td>
<td>SIMD instructions, Thumb 2, Multicore, Fast Context Switch Extension</td>
</tr>
<tr>
<td>ARM v7</td>
<td>profiles: Cortex- A (applications), -R (real-time), -M (microcontroller)</td>
</tr>
<tr>
<td>ARM v8</td>
<td>Supports 64-bit data / addressing (registers). Assembly language overview available (more than 100 pages pure instruction semantics)</td>
</tr>
</tbody>
</table>

## ARM Processor Families

very simplified & sparse

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Product Line / Family (Implementation)</th>
<th>Speed (MIPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv1-ARMv3</td>
<td>ARM1-3, 6</td>
<td>4-28 (@8-33MHz)</td>
</tr>
<tr>
<td>ARMv3</td>
<td>ARM7</td>
<td>18-56 MHz</td>
</tr>
<tr>
<td>ARMv4T, ARMv5TEJ</td>
<td>ARM7TDMI</td>
<td>up to 60</td>
</tr>
<tr>
<td>ARMv4</td>
<td>StrongARM</td>
<td>up to 200 (@200MHz)</td>
</tr>
<tr>
<td>ARMv4</td>
<td>ARM8</td>
<td>up to 84 (@72MHz)</td>
</tr>
<tr>
<td>ARMv4T</td>
<td>ARM9TDMI</td>
<td>200 (@180MHz)</td>
</tr>
<tr>
<td>ARMv5TE(J)</td>
<td>ARM9E</td>
<td>220(@200MHz)</td>
</tr>
<tr>
<td>ARMv5TE(J)</td>
<td>ARM10E</td>
<td></td>
</tr>
<tr>
<td>ARMv5TE</td>
<td>XScale</td>
<td>up to 1000 @1.25GHz</td>
</tr>
<tr>
<td>ARMv6</td>
<td>ARM11</td>
<td>740</td>
</tr>
<tr>
<td>ARMv6, ARMv7, ARMv8</td>
<td>ARM Cortex</td>
<td>up to 2000 (@&gt;1GHz)</td>
</tr>
</tbody>
</table>
ARM Architecture Reference Manuals describe

- ARM/Thumb instruction sets
- processor modes and states
- exception and interrupt model
- system programmer's model, standard coprocessor interface
- memory model, memory ordering and memory management for different potential implementations
- (optional) extensions like Floating Point, SIMD, Security, Virtualization ...

for example required for the implementation of assembler, disassembler, compiler, linker and debugger and for the systems programmer.
describe

- particular processor implementation of an ARM architecture
- redundant information from the Architecture manual (e.g. system control processor)
- additional processor implementation specifics (e.g. cache sizes and cache handling, interrupt controller, generic timer)

usually required by a system's programmer
System on Chip Implementation Manuals

describe

- particular implementation of a System on Chip
- address map:
  physical addresses and bit layout for the registers
- peripheral components / controllers,
  such as Timers, Interrupt controller, GPIO, USB, SPI, DMA, PWM, UARTs

usually required by a system's programmer.
ARM Instruction Set

consists of

- Data processing instructions
- Branch instructions
- Status register transfer instructions
- **Load and Store** instructions
- Generic Coprocessor instructions
- Exception generating instructions
Some Features
of the ARM Instruction Set

- 32 bit instructions / many in one cycle / 3 operands

- Load / store architecture (no memory operands such as in x86)

\[
\text{ldr r11, [fp, #-8]} \\
\text{add r11, r11, #1} \\
\text{str r11, [fp, #-8]}
\]
Some Features
of the ARM Instruction Set

- Index optimized instructions (such as pre-/post-indexed addressing)

```
stmdb sp!,{fp,lr} ; store multiple decrease before and update sp

ldmia sp!,{fp,pc} ; load multiple decrease after and update sp
```
Some Features
of the ARM Instruction Set

- **Predication**: all instructions can be conditionally executed*

```plaintext
cmp  r0, #0
swieq #0xa
```

null pointer check
Some Features
of the ARM Instruction Set

Link Register

- Shift and rotate in instructions

```c
add r11, fp, r11, lsl #2
```

```c
r11 = fp + r11 * 4
```

- procedure call

```c
bl #0x0a0100070
```
Some Features of the ARM Instruction Set

- **PC-relative addressing**

  ```asm
  ldr r0, [pc, #+24]
  ```

- **Coprocessor access instructions**

  ```asm
  mrc p15, 0, r11, c6, c0, 0
  ```

- **Load a large constant**

- **Setup the MMU**
### ARM Instruction Set

#### Encoding (ARM v5)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Opcode</th>
<th>S</th>
<th>Rn</th>
<th>Rd</th>
<th>Shift Amount</th>
<th>Shift</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>cond [1]</td>
<td>0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond [1]</td>
<td>0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond [1]</td>
<td>0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>cond [1]</td>
<td>0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Data processing immediate shift**
- **Shiftable register**
- **Conditional execution**
- **8 bit immediates with even rotate**
- **Load/store with destination increment**
- **Undefined instruction: user extensibility**
- **Load/store with multiple registers**
- **Branches with 24 bit offset**
- **Generic coprocessor instructions**

---

From ARM Architecture Reference Manual
Thumb Instruction Set

ARM instruction set complemented by

- Thumb Instruction Set
  - 16-bit instructions, 2 operands
  - eight GP registers accessible from most instructions
  - subset in functionality of ARM instruction set
  - targeted for density from C-code (~65% of ARM code size)

- Thumb2 Instruction Set
  - extension of Thumb, adds 32 bit instructions to support almost all of ARM ISA (different from ARM instruction set encoding!)
  - design objective: ARM performance with Thumb density
Other Contemporary RISC Architectures

Examples

- **MIPS (MIPS Technologies)**
  - Business model similar to that of ARM
  - Architectures: MIPS(I|...|V), MIPS(32|64), microMIPS(32|64)

- **AVR (Atmel)**
  - Initially targeted towards microcontrollers
  - Harvard Architecture designed and Implemented by Atmel
  - Families: tinyAVR, megaAVR, AVR32
  - AVR32: mixed 16-/32-bit encoding

- **SPARC (Sun Microsystems)**
  - Available as open-source: e.g. LEON (FPGA)

- ...
ARM from v5 has (at least) seven basic operating modes

- Each mode has access to own stack and a different subset of registers
- Some operations can only be carried out in a privileged mode

<table>
<thead>
<tr>
<th>Mode</th>
<th>Description / Cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor</td>
<td>Reset / Software Interrupt</td>
</tr>
<tr>
<td>FIQ</td>
<td>Fast Interrupt</td>
</tr>
<tr>
<td>IRQ</td>
<td>Normal Interrupt</td>
</tr>
<tr>
<td>Abort</td>
<td>Memory Access Violation</td>
</tr>
<tr>
<td>Undef</td>
<td>Undefined Instruction</td>
</tr>
<tr>
<td>System</td>
<td>Privileged Mode with same registers as in User Mode</td>
</tr>
<tr>
<td>User</td>
<td>Regular Application Mode</td>
</tr>
</tbody>
</table>
ARM Register Set

ARM has 37 registers, all 32-bits long
A subset is accessible in each mode
Register 13 is the Stack Pointer (by convention)
Register 14 is the Link Register**
Register 15 is the Program Counter (settable)
CPSR* is not immediately accessible

* current / saved processor status register, accessible via MSR / MRS instructions
** more than a convention: link register set as side effect of some instructions
# Processor Status Register (PSR)

## Condition Codes
- **N** = Negative result from ALU
- **Z** = Zero result from ALU
- **C** = ALU operation Carried out *
- **V** = ALU operation overflowed

## Interrupt Disable bits
- **I** = 1: Disables IRQ
- **F** = 1: Disables FIQ

## Mode Bits
- Specify processor mode

## Interrupt Disable bits Table

<table>
<thead>
<tr>
<th>N</th>
<th>Z</th>
<th>C</th>
<th>V</th>
<th>Q</th>
<th>J</th>
<th>GE[3:0]</th>
<th>IT cond</th>
<th>E</th>
<th>A</th>
<th>I</th>
<th>F</th>
<th>T</th>
<th>mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>28</td>
<td>27</td>
<td>24</td>
<td>23</td>
<td>20</td>
<td>19</td>
<td>16</td>
<td>15</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
</tr>
</tbody>
</table>

## Other bits
- architecture 5TE(J) and later
  - Q flag: sticky overflow flag for saturating instr.
  - J flag: Jazelle state
- architecture 6 and later
  - GE[0:3]: used by SIMD instructions
  - E: controls endianess
  - A: controls imprecise data aborts
  - IT: controls conditional execution of Thumb2

## T Bit
- **T** = 0: Processor in ARM mode
- **T** = 1: Processor in Thumb State
- Introduced in Architecture 4T

* reverse cmp/sub meaning compared with x86
## Typical procedure call on ARM

**Caller:** push parameters
- Use branch and link instruction. Stores the PC of the next instruction into the link register.

**Callee:** save link register and frame pointer on stack and set new frame pointer.
- Execute procedure content
- Reset stack pointer and restore frame pointer and jump back to caller address.

**Caller:** cleanup parameters from stack

### Instructions
- **Caller:**
  - `push parameters`
  - `use branch and link instruction. Stores the PC of the next instruction into the link register.`
  - `bl #address`
  - `stmdb sp!, {fp, lr}`
  - `mov fp, sp`
  - `...`

- **Callee:**
  - `save link register and frame pointer on stack and set new frame pointer`.
  - `mov sp, fp`
  - `ldmia sp!, {fp, pc}`
  - `...`

- **Caller:**
  - `cleanup parameters from stack`
  - `add sp, sp, #n`
  - `...`
Exceptions (General)

Exception = abrupt change in the control flow as a response to some change in the processor's state

- **Interrupt** - asynchronous event triggered by a device signal
- **Trap / Syscall** - intentional exception
- **Fault** - error condition that a handler might be able to correct
- **Abort** - error condition that cannot be corrected
Exception Handling

Involves close interaction between hardware and software.

Exception handling is similar to a procedure call with important differences:

- processor prepares exception handling: save* part of the current processor state before execution of the software exception handler
- assigned to each exception is an exception number, the exception handler's code is accessible via some exception table that is configurable by software
- exception handlers run in a different processor mode with complete access to the system resources.

* In special registers or on the stack – we will go into the details for some architectures
# Exception Table on ARM

<table>
<thead>
<tr>
<th>Type</th>
<th>Mode</th>
<th>Address*</th>
<th>return link(type)**</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>0x0</td>
<td>undef</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>Undefined</td>
<td>0x4</td>
<td>next instr</td>
</tr>
<tr>
<td>SWI</td>
<td>Supervisor</td>
<td>0x8</td>
<td>next instr</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>0xC</td>
<td>aborted instr +4</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
<td>0x10</td>
<td>aborted instr +8</td>
</tr>
<tr>
<td>Interrupt (IRQ)</td>
<td>IRQ</td>
<td>0x18</td>
<td>next instr +4</td>
</tr>
<tr>
<td>Fast Interrupt (FIQ)</td>
<td>FIRQ</td>
<td>0x1C</td>
<td>next instr +4</td>
</tr>
</tbody>
</table>

* alternatively High Vector Address = 0xFFFFF0000 + adr (configurable)
** different numbers in Thumb instruction mode
Context change, schematic

Before the interrupt

In the interrupt handler

*Processor Status Word
## Exception handling on ARM

### Hardware action at entry (invoked by exception)
- R14(exception_mode):= return link
- SPSR(exception_mode) := CPSR
- CPSR[4:0] := exception_mode number
- CPSR[5] := 0 (* execute in ARM state *)
- If exception_mode = Reset or FIQ then CPSR[6]=1 (* disable fast IRQ *)
- CPSR[7]=1 (* disable normal interrupts *)
- PC:=exception vector address

### Software
- STMDB SP!, {R0 .. R11, FP, LR} (* store all non-banked registers on stack *)
- ... (* exception handler *)
- LDMIA SP! {R0..R11,FP,LR} (* read back all non-banked registers from stack*)
- SUBS PC,LR, #ofs (* return from interrupt instruction *)

### Hardware action at exit (invoked by MOVS or SUBS instruction)
- CPSR := SPSR(exception mode) (* includes a reset of the irq/fiq flag *)
- PC := LR – ofs
Raspberry Pi 2

- Raspberry Pi 2 will be the hardware used at least in the first 4 weeks lab sessions
- Produced by element14 in the UK (www.element14.com)
- Features
  - Broadcom BCM2836 ARMv7 Quad Core Processor running at 900 MHz
  - 1G RAM
  - 40 PIN GPIO
  - Separate GPU ("Videocore")
  - Peripherals: UART, SPI, USB, 10/100 Ethernet Port (via USB), 4pin Stereo Audio, CSI camera, DSI display, Micro SD Slot
  - Powered from Micro USB port
ARM System Boot

- ARM processors usually starts executing code at adr 0x0 - e.g. containing a branch instruction to jump over the interrupt vectors - usually requires some initial setup of the hardware

- The RPI, however, is booted from the Video Core CPU (VC): the firmware of the RPI does a lot of things before we get control: kernel-image gets copied to address 0x8000H and branches there No virtual to physical address-translation takes place in the beginning.

- Only one core runs at that time. (More on this later)
RPI 1 Memory Map
Initially the MMU is switched off. No memory translation takes place.

System memory divided in ARM and VC part, partially shared (e.g. frame buffer)

ARM's memory mapped registers start from 0x3F000000 -- opposed to reported offset 0x7E000000 in BCM 2835 Manual
General Purpose I/O (GPIO)

- Software controlled processor pins
  - Configurable direction of transfer
  - Configurable connection
    - with internal controller (SPI, MMC, memory controller, ...)
    - with external device

- Pin state settable & gettable
  - High, low

- Forced interrupt on state change
  - On falling/ rising edge
GPIO
Block Diagram (BCM 2835)
Raspberry Pi 2 GPIO Pinout

<table>
<thead>
<tr>
<th>name</th>
<th>pin</th>
<th>pin</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3 V DC</td>
<td>01</td>
<td>02</td>
<td>DC power 5v</td>
</tr>
<tr>
<td>GPIO 02</td>
<td>03</td>
<td>04</td>
<td>DC power 5v</td>
</tr>
<tr>
<td>GPIO 03</td>
<td>05</td>
<td>06</td>
<td>ground</td>
</tr>
<tr>
<td>GPIO 04</td>
<td>07</td>
<td>08</td>
<td>GPIO 14</td>
</tr>
<tr>
<td>ground</td>
<td>09</td>
<td>10</td>
<td>GPIO 15</td>
</tr>
<tr>
<td>GPIO 17</td>
<td>11</td>
<td>12</td>
<td>GPIO 18</td>
</tr>
<tr>
<td>GPIO 27</td>
<td>13</td>
<td>14</td>
<td>ground</td>
</tr>
<tr>
<td>GPIO 22</td>
<td>15</td>
<td>16</td>
<td>GPIO 23</td>
</tr>
<tr>
<td>3.3V DC</td>
<td>17</td>
<td>18</td>
<td>GPIO 24</td>
</tr>
<tr>
<td>GPIO 10</td>
<td>19</td>
<td>20</td>
<td>ground</td>
</tr>
<tr>
<td>GPIO 09</td>
<td>21</td>
<td>22</td>
<td>GPIO 25</td>
</tr>
<tr>
<td>GPIO 11</td>
<td>23</td>
<td>24</td>
<td>GPIO 08</td>
</tr>
<tr>
<td>ground</td>
<td>25</td>
<td>26</td>
<td>GPIO 07</td>
</tr>
<tr>
<td>ID_SD</td>
<td>27</td>
<td>28</td>
<td>ID_SC</td>
</tr>
<tr>
<td>GPIO 05</td>
<td>29</td>
<td>30</td>
<td>ground</td>
</tr>
<tr>
<td>GPIO 06</td>
<td>31</td>
<td>32</td>
<td>GPIO 12</td>
</tr>
<tr>
<td>GPIO 13</td>
<td>33</td>
<td>34</td>
<td>ground</td>
</tr>
<tr>
<td>GPIO 19</td>
<td>35</td>
<td>36</td>
<td>GPIO 16</td>
</tr>
<tr>
<td>GPIO 26</td>
<td>37</td>
<td>38</td>
<td>GPIO 20</td>
</tr>
<tr>
<td>ground</td>
<td>39</td>
<td>40</td>
<td>GPIO 21</td>
</tr>
</tbody>
</table>
## Documentation Examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Field Name</th>
<th>Description</th>
<th>Size</th>
<th>Read/Write</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 7E20 0000</td>
<td>GPFSFL0</td>
<td>GPIO Function Select 0</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0000</td>
<td>GPFSFL0</td>
<td>GPIO Function Select 0</td>
<td>32</td>
<td>R/W</td>
</tr>
<tr>
<td>0x 7E20 0004</td>
<td>GPFSFL1</td>
<td>GPIO Function Select 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x 7E20 0008</td>
<td>GPFSFL2</td>
<td>GPIO Function Select 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x 7E20 000C</td>
<td>GPFSFL3</td>
<td>GPIO Function Select 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0x 7E20 0010</td>
<td>GPFSFL4</td>
<td>GPIO Function Select 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Bit(s) | Field Name | Description                                                                 | Type | Reset |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31-30</td>
<td>---</td>
<td>Reserved</td>
<td>R</td>
<td>0</td>
</tr>
<tr>
<td>29-27</td>
<td>FSEL19</td>
<td>FSEL19 - Function Select 19</td>
<td>RW</td>
<td>0</td>
</tr>
<tr>
<td>26-24</td>
<td>FSEL18</td>
<td>FSEL18 - Function Select 18</td>
<td>RW</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6-3 – GPIO Alternate function select register 1
GPIO Setup (RPI2)

1. Program GPIO Pin Function (in / out / alternate function) 
   by writing corresponding (memory mapped) GPFSEL register. 
   GPFSELn: pins 10*n .. 10*n+9
   Use RMW (Read-Modify-Write) operation in order to keep the other bits

2. Use GPIO Pin
   a. If writing: set corresponding bit in the GPSETn or GPCLRn register
      set pin: GPSETn: pins 32*n .. 32*n+31
      clear pin: GPCLRn: pins 32*n .. 32*n+31
      no RMW required.
   b. If reading: read corresponding bit in the GPLEVn register 
      GPLEVn: pins 32*n ... 32*n+1
   c. If "alternate function": device acts autonomously. Implement device driver.