18-349: Embedded Real-Time Systems
Lecture 2: ARM Architecture

Anthony Rowe
Electrical and Computer Engineering
Carnegie Mellon University
Basic Computer Architecture

![Diagram of computer architecture](image-url)

- **Central Processing Unit (CPU)**
  - Control Unit
  - ALU
  - Registers
  - Main Memory
  - Secondary Memory
  - Storage

- **Bus**
  - Keyboard
  - Mouse
  - Input Devices
  - Display
  - Printer
  - Output Devices

![Memory Hierarchy Triangle](image-url)

- **CPU Registers**
- **Level 1 Cache**
- **Level 2 Cache**
- **Physical RAM (Main Memory)**
- **Disc Storage (Virtual RAM, Hard Drive)**

**Capacity**

**Memory Hierarchy**

**Cost per byte, access speed**

---

Electrical & Computer Engineering

Carnegie Mellon University
Memory Types

- **DRAM: Dynamic Random Access Memory**
  - Upside: very dense (1 transistor per bit) and cheap
  - Downside: requires refresh and often slow
  - Used as main memory

- **SRAM: Static Random Access Memory**
  - Upside: fast and no refresh required
  - Downside: not so dense, not so cheap
  - Often used for caches

- **EEPROM: Electronically Erasable Programmable Read-only Memory**
  - Used for bootstrapping
  - Require wear-leveling
Big Endian vs. Little Endian

- How is a word, say, 0x1234567 stored in memory?

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x103</td>
<td>78</td>
</tr>
<tr>
<td>0x102</td>
<td>56</td>
</tr>
<tr>
<td>0x101</td>
<td>34</td>
</tr>
<tr>
<td>0x100</td>
<td>12</td>
</tr>
</tbody>
</table>

Big Endian

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x103</td>
<td>12</td>
</tr>
<tr>
<td>0x102</td>
<td>34</td>
</tr>
<tr>
<td>0x101</td>
<td>56</td>
</tr>
<tr>
<td>0x100</td>
<td>78</td>
</tr>
</tbody>
</table>

Little Endian
Big Endian vs. Little Endian

- **Big-endian (big end first)**
  - Most significant byte of any multi-byte data field is stored at the lowest memory address
  - Reading from left to right
  - SPARC & Motorola

- **Little-endian (little end first)**
  - Least significant byte of any multi-byte data field is stored at the lowest memory address
  - Reading from right to left instead
  - Intel processors

- **Bi-endian (ARM, PowerPC, Alpha)**
CISC vs. RISC

- RISC – Reduced Instruction Set Computers
- CISC – Complex Instruction Set Computers
- Different architectures for doing the same operations
- Suppose you wanted to multiply two numbers in memory locations \textit{mem0} & \textit{mem1} and store the results back in \textit{mem0}
  - Same result but the complexity of operations and the number of steps used in the two cases differ

<table>
<thead>
<tr>
<th>CISC Approach</th>
<th>RISC Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mull mem0, mem1}</td>
<td>\texttt{ldr r0, mem0}</td>
</tr>
<tr>
<td></td>
<td>\texttt{ldr r1, mem1}</td>
</tr>
<tr>
<td></td>
<td>\texttt{mul r0, r0, r1}</td>
</tr>
<tr>
<td></td>
<td>\texttt{str mem0, r0}</td>
</tr>
</tbody>
</table>
## CISC vs. RISC (1)

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example – Intel x86 chips</td>
<td>Examples – SPARC, PowerPC, ARM</td>
</tr>
<tr>
<td>Large number of instructions</td>
<td>Few instructions, typically less than 100</td>
</tr>
<tr>
<td>Variable-length instructions, instructions can range from 1-15 bytes</td>
<td>Fixed-length instructions, all instructions have the same number of bytes</td>
</tr>
<tr>
<td>Some instructions can have long execution times</td>
<td>No instruction with a long execution times execution time</td>
</tr>
</tbody>
</table>
## CISC vs. RISC (2)

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
</table>
| Arithmetic and logical operations can be applied to memory and register operands | Arithmetic and logical operations only use register operands  
  - Memory contents have to be loaded into registers first  
  - Referred to as load/store architecture |
| Stack-intensive procedure linkage  
  - Stack is used for procedure arguments and return values | Register-intensive procedure linkage  
  - Registers used for procedure arguments and return values |
## CISC vs. RISC (3)

<table>
<thead>
<tr>
<th>CISC</th>
<th>RISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact code size is typically small</td>
<td>Compiled code size is larger</td>
</tr>
<tr>
<td>More transistors = more power</td>
<td>Fewer transistors = less power</td>
</tr>
</tbody>
</table>
ARM, Ltd.

- Founded in November 1990
  - Spun out of Acorn Computers based in U.K.
  - ARM was originally Acorn RISC Machine; then, Advanced RISC Machine

- Most widely used 32-bit instruction-set architecture in terms of volume
  - 6.1 billion ARM processors in 2010 up to 15 billion in 2015
  - 95% of all smartphones, 35% of digital TVs and set-top boxes

- ARM architecture can be licensed, with licensees (former and/or current)
  - AMD, Apple, Freescale, Microsoft, Nintendo, Xilinx, Qualcomm, TI, etc.

- Companies design custom CPU cores with ARM instruction set
  - Qualcomm’s Snapdragon, Apple’s A8, etc.

- ARM Ltd. does not fabricate processors itself
  - Also develops technologies to help with the design-in of ARM devices
  - Software tools, boards, debug hardware, application software, buses, peripherals
## History of ARM’s Usage

<table>
<thead>
<tr>
<th>Example ARM component</th>
<th>Architecture Generation</th>
<th>Example Application</th>
<th>Approximate date of introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM1</td>
<td>ARMv1</td>
<td>Acorn Computer in internal testing</td>
<td>1985</td>
</tr>
<tr>
<td>ARM2</td>
<td>ARMv2</td>
<td>Acorn Archimedes (Macintosh-era PC)</td>
<td>1987</td>
</tr>
<tr>
<td>ARM6</td>
<td>ARMv3</td>
<td>Apple Newton MessagePad 100 series</td>
<td>1994</td>
</tr>
<tr>
<td>ARM7TDMI</td>
<td>ARMv4</td>
<td>Game Boy Advance, Nintendo DS*, iPod</td>
<td>2001</td>
</tr>
<tr>
<td>ARM9E</td>
<td>ARMv5</td>
<td>Nintendo DS*, Nokia N-Gage, Airport Extreme N basestation</td>
<td>2004</td>
</tr>
<tr>
<td>ARM11</td>
<td>ARMv6</td>
<td>iPhone, iPhone 3G, iPod touch</td>
<td>2007</td>
</tr>
<tr>
<td>Cortex-A8</td>
<td>ARMv7</td>
<td>Palm Pre, iPhone 3GS</td>
<td>2009</td>
</tr>
</tbody>
</table>
SoftBank Acquisition

- Japanese multinational company
- July 18th 2016
- Purchased for £23.4 billion
- Speculating on IoT market
## ARM Everywhere

<table>
<thead>
<tr>
<th>Industry</th>
<th>Market Size</th>
<th>Growth Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile</td>
<td>4.6bn</td>
<td>3% in 2012</td>
</tr>
<tr>
<td>Embedded</td>
<td>2.3bn</td>
<td>25% in 2012</td>
</tr>
<tr>
<td>Enterprise</td>
<td>1.4bn</td>
<td>10% in 2012</td>
</tr>
<tr>
<td>Home</td>
<td>0.4bn</td>
<td>40% in 2012</td>
</tr>
</tbody>
</table>

Electrical & Computer Engineering

Carnegie Mellon University
The ARM Family

- Classic ARM Processors
  - ARM7
  - ARM9
  - ARM11

- Embedded Cortex Processors
  - Cortex-M0
  - Cortex-M1
  - Cortex-M3
  - Cortex-M4

- Application Cortex Processors
  - Cortex-A5
  - Cortex-A8
  - Cortex-A9
  - Cortex-A15

Performance Functionality vs. Capability
Different ARM Core Families

- **Cortex-A**
  - Application processors
  - Single-core or multi-core
  - Optional multimedia processing
  - Optional floating-point units
  - Smartphones, tablets, digital TVs, eBook readers

- **Cortex-R**
  - Deeply embedded real-time applications
  - Low power, good interrupt behavior with good performance
  - Automotive braking systems, printers, storage controllers

- **Cortex-M**
  - Cost-sensitive microcontrollers
  - Fast, deterministic interrupt management
  - Lowest possible power consumption
  - Automotive airbags, tire-pressure monitoring, smart meters, sensors
Cortex-M Family

Cortex-M0
- Lowest cost
- Outstanding energy efficiency
- Low area

Cortex-M0+
- 90 μm
- 15 years

Cortex-M3
- Performance efficiency
- Feature rich connectivity

Cortex-M4
- Digital Signal Control (DSC) Processor with DSP
- Accelerated SIMD
- Floating point (FP)

Cortex-M7
- Maximum DSC Performance
- Flexible Memory System
- Cache, TCM, AXI, ECC
- Double & Single Precision FP

Scalable and Compatible Architecture

Digital Signal Control application space

‘8/16-bit’ Traditional application space

‘16/32-bit’ Traditional application space
Beyond Cortex

Different processors optimized for different tasks

Cortex-A7 manages always-on, always connected workloads

Cortex-A15 handles peak performance tasks

Social Media

Video
ARM Data Sizes & Instructions

- The ARM is a 32-bit RISC architecture

- When used in relation to the ARM
  - **Byte** means 8-bits
  - **Halfword** means 16 bits (two bytes)
  - **Word** means 32 bits (four bytes)

- Most ARM processors implement two instructions sets
  - 32-bit ARM Instructions Set
  - 16-bit Thumb Instruction Set

- Bi-endian
  - Can be configured to view words stored in memory as either Big-endian or Little-Endian Format
ARM is a RISC Architecture

- A large array of uniform registers

- A load/store model, where
  - Operations operate only on register and not directly on memory
  - All data must be loaded into registers before being used
  - Result (in a register) can be further processed or stored to memory

- A small number of addressing modes
  - All load / store addresses are determined from register and instruction fields

- A uniform fixed-length instruction (32-bit)
Programmer’s Model

- ARM supports seven processor modes
  - Characterized by specific behavior, privileges, associated registers
  - Mode changes can be made under software control, or be caused by external interrupts or exception processing

- Most applications execute in **User mode**
  - Program cannot access certain protected resources
  - Program cannot change mode without causing an exception

- The 6 modes other than user mode are called **privileged modes**
  - 5 of these privileged modes are called **exception modes**
  - The remaining one is called the **System mode** (same as User mode, but with access to protected resources)
## The Seven Modes

<table>
<thead>
<tr>
<th>Processor mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>User</td>
<td>Normal program execution mode</td>
</tr>
<tr>
<td>FIQ</td>
<td>Fast Interrupt for high-speed data transfer</td>
</tr>
<tr>
<td>IRQ</td>
<td>Used for general-purpose interrupt handling</td>
</tr>
<tr>
<td>Supervisor</td>
<td>A protected mode for the operating system</td>
</tr>
<tr>
<td>Abort</td>
<td>Implements virtual memory and/or memory protection</td>
</tr>
<tr>
<td>Undefined</td>
<td>Supports software emulation of hardware coprocessors</td>
</tr>
<tr>
<td>System</td>
<td>Runs privileged operating system tasks</td>
</tr>
</tbody>
</table>
Register Set (1)

- ARM has a total of 37 registers all of which are 32-bit
  - 30 general-purpose registers
  - 1 dedicated program counter (pc)
  - 1 dedicated current program status register (cpsr)
  - 5 dedicated saved program status registers (spsr)

- In any mode, only a subset of these 37 registers are visible
  - The hidden registers are called **banked registers**
  - The current processor-mode governs which registers are visible
Register Set (2)

- **r0 through r7**: Eight general-purpose registers that are always available, no matter which mode you’re in (8)

- **r8 through r12**: Five general-purpose registers that are common to all processor modes other than fiq mode (5)

- **r8_fiq through r12_fiq**: Five registers that replace the normal r8-r12 when the processor is in fiq mode (5)

- **Special-purpose registers**
  - **r13 (stack pointer)**: Same for System and User mode, otherwise, r13_fiq, r13_svc, r13_abt, r13_irq, r13_und (6)
  - **r14 (link register)**: Same for System and User mode, otherwise, r14_fiq, r14_svc, r14_abt, r14_irq, r14_und (6)
  - **r15 (program counter)**: A unique one across all modes (1)
Register Set (3)

- Status registers
  - **cpsr (current program status register):** Holds current status of processor, including its mode (1)
  - **spsr (saved program status register):** Holds processor status information before program changes into an exception mode, r13_fiq, r13_svc, r13_abt, r13_irq, r13_und (5)
## Register Set (4)

<table>
<thead>
<tr>
<th>System &amp; User</th>
<th>FIQ</th>
<th>Supervisor</th>
<th>Abort</th>
<th>IRQ</th>
<th>Undefined</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
<td>R8_fiq</td>
<td>14</td>
<td>R8</td>
<td>R8</td>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
<td>R9_fiq</td>
<td>15</td>
<td>R9</td>
<td>R9</td>
<td>R9</td>
</tr>
<tr>
<td>R10</td>
<td>R10_fiq</td>
<td>16</td>
<td>R10</td>
<td>R10</td>
<td>R10</td>
</tr>
<tr>
<td>R11</td>
<td>R11_fiq</td>
<td>17</td>
<td>R11</td>
<td>R11</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>R12_fiq</td>
<td>18</td>
<td>R12</td>
<td>R12</td>
<td>R12</td>
</tr>
<tr>
<td>R13</td>
<td>R13_fiq</td>
<td>19</td>
<td>R13_svc</td>
<td>23</td>
<td>R13_svc</td>
</tr>
<tr>
<td>R14</td>
<td>R14_fiq</td>
<td>20</td>
<td>R14_svc</td>
<td>24</td>
<td>R14_svc</td>
</tr>
<tr>
<td>R15(PC)</td>
<td>R15(PC)</td>
<td>21</td>
<td>R13_abt</td>
<td>25</td>
<td>R13_abt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R13_irq</td>
<td>27</td>
<td>R13_irq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R13_svc</td>
<td>28</td>
<td>R13_svc</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R13_abt</td>
<td>29</td>
<td>R13_abt</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R13_irq</td>
<td>30</td>
<td>R13_irq</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R13_svc</td>
<td>31</td>
<td>R13_svc</td>
</tr>
</tbody>
</table>

### ARM State Program Status Registers

- CPSR
- SPSR_fiq
- SPSR_svc
- SPSR_abt
- SPSR_irq
- SPSR_und

= banked register
Banked Registers

- Banking of registers implies
  - The specific register depends not only on the number (r0, r1, …, r15) but also on the processor mode

- Values stored in banked registers are preserved across mode changes

Example: Assume that the processor is executing in User Mode
  - In **User** mode, assume that the processor writes 0 in r0 and 8 in r8
  - Processor now changes to **fiq** mode
    - In FIQ mode, the value of r0 is __________
    - If processor now overwrites both r0 and r8 with 1 in **fiq** mode and changes back to **user** mode
      - The new value stored in r0 (user mode) is __________
      - The new value stored in r8 (user mode) is __________
Register Set in User Mode

Current Visible Registers

User Mode

- r0
- r1
- r2
- r3
- r4
- r5
- r6
- r7
- r8
- r9
- r10
- r11
- r12
- r13 (sp)
- r14 (lr)
- r15 (pc)
- cpsr

Banked out Registers

- FIQ: r8, r9, r10, r11, r12, r13 (sp), r14 (lr)
- IRQ: r13 (sp), r14 (lr)
- SVC: r13 (sp), r14 (lr)
- Undef: r13 (sp), r14 (lr)
- Abort: r13 (sp), r14 (lr)
Register Set in \texttt{fiq} Mode

Current Visible Registers

FIQ Mode

- \texttt{r0}
- \texttt{r1}
- \texttt{r2}
- \texttt{r3}
- \texttt{r4}
- \texttt{r5}
- \texttt{r6}
- \texttt{r7}
- \texttt{r8}
- \texttt{r9}
- \texttt{r10}
- \texttt{r11}
- \texttt{r12}
- \texttt{r13 (sp)}
- \texttt{r14 (lr)}
- \texttt{r15 (pc)}
- \texttt{cpsr}
- \texttt{spsr}

Banked out Registers

- \texttt{r8}
- \texttt{r9}
- \texttt{r10}
- \texttt{r11}
- \texttt{r12}
- \texttt{r13 (sp)}
- \texttt{r14 (lr)}

<table>
<thead>
<tr>
<th>User</th>
<th>IRQ</th>
<th>SVC</th>
<th>Undef</th>
<th>Abort</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{r13 (sp)}</td>
<td>\texttt{r13 (sp)}</td>
<td>\texttt{r13 (sp)}</td>
<td>\texttt{r13 (sp)}</td>
<td>\texttt{r13 (sp)}</td>
</tr>
<tr>
<td>\texttt{r14 (lr)}</td>
<td>\texttt{r14 (lr)}</td>
<td>\texttt{r14 (lr)}</td>
<td>\texttt{r14 (lr)}</td>
<td>\texttt{r14 (lr)}</td>
</tr>
</tbody>
</table>

- \texttt{spsr}
- \texttt{spsr}
- \texttt{spsr}
- \texttt{spsr}
Thumb Mode

- Thumb is a 16-bit instruction set
  - Optimized for code density from C code (~65% of ARM code size)
  - Improved performance from narrow memory
  - Subset of the functionality of the ARM instruction set

- Core has additional execution state – Thumb
  - Switch between ARM and Thumb using BX instruction

- For most instructions generated by compiler:
  - Conditional execution is not used
  - Source and destination registers identical
  - Only Low registers used
  - Constants are of limited size
  - Inline barrel shifter not used
The CPSR (1)

- Current Program Status Register (cpsr) is a dedicated register
- Holds information about the most recently performed ALU operations
- Controls the enabling and disabling of interrupts (both IRQ and FIQ)
- Sets the processor operating mode
- Sets the processor state
The CPSR (2)

- cpsr has two important pieces of information
  - Flags: contains the condition flags
  - Control: contains the processor mode, state and interrupt mask bits

- All fields of the cpsr can be read/written in privileged modes
  - Only the flag field of cpsr can be written in User mode, all fields can be read in User mode
The CPSR (3)

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

- **T Bit**
  - $T = 0$: Processor in ARM state
  - $T = 1$: Processor in Thumb state

- **Mode bits**
  - Specify the processor mode

When exceptions occur, cpsr gets copied to the corresponding spsr_<mode> register for storage
The CPSR (4)

- Will represent this as nzcvqift_mode
- Upper case letters will indicate that a certain bit has been set

Examples
- nzcvqiFt_USER: FIQs are masked and the processor is executing in user mode
- nzCvqift_SVC: Carry flag is set and the processor is executing in supervisor mode
Exceptions vs. Interrupts

- Term exception and interrupt are often confused!

- Exception usually refers to an internal CPU event such as
  - Floating point overflow
  - MMU fault (e.g., page fault)
  - Trap (SWI)

- *Interrupt* usually refers to an external I/O event such as
  - I/O device request
  - Reset

- In the ARM architecture manuals, the two terms *are* mixed together and are considered interchangeable
## ARM Exceptions

<table>
<thead>
<tr>
<th>Exception</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reset</td>
<td>Supervisor</td>
<td>Occurs when the processor’s reset button is asserted. This exception is only expected to occur for signaling power up or for resetting the processor. A soft reset can be achieved by branching to reset vector 0x00000000 or letting the watchdog timer expire.</td>
</tr>
<tr>
<td>Undefined Instruction</td>
<td>Undef</td>
<td>Occurs if neither the processor, nor any of the coprocessors, recognize the currently executing instruction.</td>
</tr>
<tr>
<td>Software Interrupt</td>
<td>Supervisor</td>
<td>This is a user-defined synchronous interrupt. It allows a program running in the User mode to request privileged operations (for example an RTOS function) that run in Supervisor mode.</td>
</tr>
<tr>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>Occurs when a processor attempts to execute an instruction that was not fetched, because the address was illegal.</td>
</tr>
<tr>
<td>Data Abort</td>
<td>Abort</td>
<td>Occurs when a data transfer instruction attempts to load or store data at an illegal address.</td>
</tr>
<tr>
<td>IRQ</td>
<td>IRQ</td>
<td>Occurs when the processor’s external interrupt request pin is asserted and the I bit in the cpsr is clear.</td>
</tr>
<tr>
<td>FIQ</td>
<td>FIQ</td>
<td>Occurs when the processor’s external fast interrupt request pin is asserted and the I (F-bit?) bit in the cpsr is clear.</td>
</tr>
</tbody>
</table>
ARM Exception Handling (1)

- Exception Handler
  - Most exceptions have an associated software exception handler that executes when that particular exception occurs

- Exception modes and registers
  - Handling exceptions changes program from user to non-user mode
  - Each exception handler has access to its own set of registers
    - Its own r13 (stack pointer)
    - Its own r14 (link register)
    - Its own spsr (Saved Program Status Register)
  - Exception handlers must save (restore) other registers on entry (exit)
ARM Exception Handling (2)

- Where is the is exception handler located?

- Vector table
  - Reserved area of 32 bytes at the end of the memory map (starting at address 0x0)
  - One word of space for each exception type
  - Contains a Branch or Load PC instruction for the exception handler
ARM Exception Handling (3)

- When an exception occurs, **the ARM processor:**
  - Copies cpsr into spsr_<mode>
  - Sets appropriate cpsr bits
    - Change to ARM state
    - Change to exception mode
    - Disable interrupts (if appropriate)
  - Stores the return address in lr_<mode>
  - Sets pc to vector address

- To return, **exception handler needs to:**
  - Restore cpsr from the spsr_<mode>
  - Restore pc from lr_<mode>
## Simultaneous Exceptions?

<table>
<thead>
<tr>
<th>Vector address</th>
<th>Exception type</th>
<th>Exception mode</th>
<th>Priority (1=high, 6=low)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>Reset</td>
<td>Supervisor (SVC)</td>
<td>1</td>
</tr>
<tr>
<td>0x4</td>
<td>Undefined Instruction</td>
<td>Undef</td>
<td>6</td>
</tr>
<tr>
<td>0x8</td>
<td>Software Interrupt (SWI)</td>
<td>Supervisor (SVC)</td>
<td>6</td>
</tr>
<tr>
<td>0xC</td>
<td>Prefetch Abort</td>
<td>Abort</td>
<td>5</td>
</tr>
<tr>
<td>0x10</td>
<td>Data Abort</td>
<td>Abort</td>
<td>2</td>
</tr>
<tr>
<td>0x14</td>
<td>Reserved</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>0x18</td>
<td>Interrupt (IRQ)</td>
<td>Interrupt (IRQ)</td>
<td>4</td>
</tr>
<tr>
<td>0x1C</td>
<td>Fast Interrupt (FIQ)</td>
<td>Fast Interrupt (FIQ)</td>
<td>3</td>
</tr>
</tbody>
</table>
Why Exceptions?

- Functionality that would otherwise not be possible
  - Memory or Data Abort can be used to implement Virtual Memory

- SWI allows for system calls

- Undefined exceptions can be used to provide software emulation of coprocessor when the coprocessor is not physically present or could be used for special purpose instruction set extensions
  - If an unknown instruction is reached the processor changes to **Undefined** mode and executes the Undefined Instruction exception handler
  - In the exception handler, the coprocessor functionality can be provided in software (or the functionality provided by the enhanced instructions can be provided in software)
Summary

- What is an ARM?
  - CISC vs RISC
  - ARM family
  - Memory

- ARM Architecture from a programmer’s viewpoint
  - Processor modes
  - General purpose registers
  - Special purpose registers
  - Exception handling

- Next Time: Deeper into ARM ASM