Lecture 3: ISA, Assembly, and Toolchains
What distinguishes embedded systems?
- Application-specific
- Resource-constrained
- Real-time operations
- Software runs “forever”
• What distinguishes embedded systems?
  - Application-specific
  - Resource-constrained
  - Real-time operations
  - Software runs “forever”

• Technology scaling is driving “embedded everywhere”
  - Microprocessors
  - Memory (RAM and Flash)
  - Imagers (i.e. camera) and MEMS sensors (e.g. accelerometer)
  - Energy storage/generation
Exercise: We study FPGAs and MPU-32. Why?
Architecture
In the context of computers, what does *architecture* mean?
Architecture has many meanings
Architecture has many meanings

- Computer Organization (or Microarchitecture)
  - Control and data paths
  - Pipeline design
  - Cache design
  - ...

...
Architecture has many meanings

- **Computer Organization (or Microarchitecture)**
  - Control and data paths
  - Pipeline design
  - Cache design
  - ...

- **System Design (or Platform Architecture)**
  - Memory and I/O buses
  - Memory controllers
  - Direct memory access
  - ...
Architecture has many meanings

- **Computer Organization (or Microarchitecture)**
  - Control and data paths
  - Pipeline design
  - Cache design
  - ...

- **System Design (or Platform Architecture)**
  - Memory and I/O buses
  - Memory controllers
  - Direct memory access
  - ...

- **Instruction Set Architecture (ISA)**
What is an *Instruction Set Architecture (ISA)*?
“Instruction set architecture (ISA) is the structure of a computer that a machine language programmer (or a compiler) must understand to write a correct (timing independent) program for that machine.”

IBM introducing 360 in 1964
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td></td>
</tr>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
</tr>
<tr>
<td>R12</td>
<td></td>
</tr>
<tr>
<td>R13 (SP)</td>
<td></td>
</tr>
<tr>
<td>R14 (LR)</td>
<td></td>
</tr>
<tr>
<td>R15 (PC)</td>
<td></td>
</tr>
<tr>
<td>xPSR</td>
<td></td>
</tr>
</tbody>
</table>
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)
**Major elements of an Instruction Set Architecture**
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

<table>
<thead>
<tr>
<th>32-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
</tr>
<tr>
<td>R10</td>
</tr>
<tr>
<td>R11</td>
</tr>
<tr>
<td><strong>R12</strong></td>
</tr>
<tr>
<td><strong>R13 (SP)</strong></td>
</tr>
<tr>
<td><strong>R14 (LR)</strong></td>
</tr>
<tr>
<td><strong>R15 (PC)</strong></td>
</tr>
<tr>
<td>xPSR</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>32-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
</tr>
<tr>
<td>0xFFFF0000</td>
</tr>
<tr>
<td>Private peripheral bus - External</td>
</tr>
<tr>
<td>0xE0100000</td>
</tr>
<tr>
<td>Private peripheral bus - Internal</td>
</tr>
<tr>
<td>0xE0040000</td>
</tr>
<tr>
<td>External device</td>
</tr>
<tr>
<td>1.0GB</td>
</tr>
<tr>
<td>0xA0000000</td>
</tr>
<tr>
<td>External RAM</td>
</tr>
<tr>
<td>1.0GB</td>
</tr>
<tr>
<td>0x60000000</td>
</tr>
<tr>
<td>Peripheral</td>
</tr>
<tr>
<td>0.5GB</td>
</tr>
<tr>
<td>0x40000000</td>
</tr>
<tr>
<td>SRAM</td>
</tr>
<tr>
<td>0.5GB</td>
</tr>
<tr>
<td>0x20000000</td>
</tr>
<tr>
<td>Code</td>
</tr>
<tr>
<td>0.5GB</td>
</tr>
<tr>
<td>0x00000000</td>
</tr>
</tbody>
</table>
### Major elements of an Instruction Set Architecture

(registers, memory, word size, endianess, conditions, instructions, addressing modes)

#### 32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>R13 (SP)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>R14 (LR)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td><strong>R15 (PC)</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xPSR</td>
</tr>
</tbody>
</table>

#### Endianess

<table>
<thead>
<tr>
<th>Address</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xFFFFFFFF</td>
<td>System</td>
</tr>
<tr>
<td>0xE0100000</td>
<td>Private peripheral bus - External</td>
</tr>
<tr>
<td>0xE0040000</td>
<td>Private peripheral bus - Internal</td>
</tr>
<tr>
<td>0xE0000000</td>
<td>External device 1.0GB</td>
</tr>
<tr>
<td>0xA0000000</td>
<td>External RAM 1.0GB</td>
</tr>
<tr>
<td>0x60000000</td>
<td>Peripheral 0.5GB</td>
</tr>
<tr>
<td>0x40000000</td>
<td>SRAM 0.5GB</td>
</tr>
<tr>
<td>0x20000000</td>
<td>Code 0.5GB</td>
</tr>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
</tr>
</thead>
<tbody>
<tr>
<td>R6</td>
<td>R7</td>
<td>R8</td>
<td>R9</td>
<td>R10</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>R13(SP)</td>
<td>R14(LR)</td>
<td>R15(PC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Endianess

32-bits

<table>
<thead>
<tr>
<th>System</th>
<th>0xFFFFFFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private peripheral bus - External</td>
<td>0xE0100000</td>
</tr>
<tr>
<td>Private peripheral bus - Internal</td>
<td>0xE0040000</td>
</tr>
<tr>
<td>0xE0000000</td>
<td></td>
</tr>
<tr>
<td>External device</td>
<td>1.0GB</td>
</tr>
<tr>
<td>External RAM</td>
<td>1.0GB</td>
</tr>
<tr>
<td>0xA0000000</td>
<td></td>
</tr>
<tr>
<td>Peripheral</td>
<td>0.5GB</td>
</tr>
<tr>
<td>0x60000000</td>
<td></td>
</tr>
<tr>
<td>SRAM</td>
<td>0.5GB</td>
</tr>
<tr>
<td>0x40000000</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>0.5GB</td>
</tr>
<tr>
<td>0x20000000</td>
<td></td>
</tr>
<tr>
<td>0x00000000</td>
<td></td>
</tr>
</tbody>
</table>
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>R14 (LR)</td>
<td>R15 (PC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>xPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Endianness

Endianness
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td>R13 (SP)</td>
<td>R14 (LR)</td>
<td>R15 (PC)</td>
<td>xPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

mov r0, #1

ld r1, [r0,#5]

mem((r0)+5)

bne loop

subs r2, #1
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)

xPSR

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianness

Endianess
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianness

32-bits

Endianess
Major elements of an Instruction Set Architecture

(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
xPSR

mov r0, #1

ld r1, [r0,#5]

mem((r0)+5)

bne loop

subs r2, #1

Endianness

32-bits

Endianness
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R13 (SP)</td>
<td>R14 (LR)</td>
<td>R15 (PC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

xPSR

Endianess

32-bits

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianess
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)

mov r0, #1

dl r1, [r0,#5]

cmp ((r0)+5)

bne loop

subs r2, #1

Endianess

Endianess
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R13 (SP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R14 (LR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R15 (PC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xPSR</td>
</tr>
</tbody>
</table>

32-bits

Endianess

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianess
An ISA defines the hardware/software interface

- A “contract” between architects and programmers
- Register set
- Instruction set
  - Addressing modes
  - Word size
  - Data formats
  - Operating modes
  - Condition codes
- **Calling conventions**
  - Really not part of the ISA (usually)
  - Rather part of the ABI (Application Binary Interface)
  - But the ISA often provides meaningful support.
ARM Architecture roadmap

4T
- ARM7TDMI
- ARM922T
  - Thumb instruction set

5TE
- ARM926EJ-S
- ARM946E-S
- ARM966E-S
  - Improved
  - ARM/Thumb Interworking
  - DSP instructions
  - Extensions: Jazelle (5TEJ)

6
- ARM1136JF-S
- ARM1176JZF-S
- ARM11 MPCore
  - SIMD Instructions
  - Unaligned data support
  - Extensions: Thumb-2 (6T2)
  - TrustZone (6Z)
  - Multicore (6K)

7
- Cortex-A8/R4/M3/M1
  - Thumb-2
  - Extensions:
    - v7A (applications) – NEON
    - v7R (real time) – HW Divide
    - V7M (microcontroller) – HW Divide and Thumb-2 only
Latest ones

Cortex-A
- Cortex-A57
- Cortex-A53
- Cortex-A15
- Cortex-A9
- Cortex-A8
- Cortex-A7
- Cortex-A5

Cortex-R
- Cortex-R7
- Cortex-R5
- Cortex-R4

Cortex-M
- Cortex-M4
- Cortex-M3
- Cortex-M1
- Cortex-M0+
- Cortex-M0

SecurCore
- SC000
- SC100
- SC300
Latest ones

- Cortex-A
  - Cortex-A57
  - Cortex-A53
  - Cortex-A15
  - Cortex-A9
  - Cortex-A8
  - Cortex-A7
  - Cortex-A5
- Cortex-R
  - Cortex-R7
  - Cortex-R5
  - Cortex-R4
- Cortex-M
  - Cortex-M4
- SecurCore
  - SC000
  - SC100
  - SC300
ARM Cortex-M3 ISA

Instruction Set

ADD Rd, Rn, <op2>

Register Set

32-bits

Address Space

32-bits

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

32-bits

Endianness

Endianness
Mode dependent

Registers

low registers

high registers

Program Status Register

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
xPSR

SP_process
SP_main

Mode dependent
Address Space

- ROM Table
- External PPB
- ETM
- TPIU
- Reserved
- SCS
- Reserved
- FPB
- DWT
- ITM
- System
- Private peripheral bus - External
- Private peripheral bus - Internal
- External device 1.0GB
- External RAM 1.0GB
- Peripheral 0.5GB
- SRAM 0.5GB
- Code 0.5GB
- 32MB Bit band alias
- 31MB
- 1MB Bit band region
- 32MB Bit band alias
- 31MB
- 1MB Bit band region
### Thumb Instruction Details

**A6.7.3 ADD (immediate)**

This instruction adds an immediate value to a register value, and writes the result to the destination register.

**Encoding T4**

<table>
<thead>
<tr>
<th>ADDW&lt;IE&gt; Rd,Rn,#imm12</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001110 iimm3 0 1 1 1 0</td>
</tr>
</tbody>
</table>

**Encoding T3**

<table>
<thead>
<tr>
<th>ADDc{S}W Rd,Rn,#const</th>
</tr>
</thead>
<tbody>
<tr>
<td>11110i01000S 0 1 1 1 0</td>
</tr>
</tbody>
</table>

**Encoding T2**

<table>
<thead>
<tr>
<th>ADDS Rd,Rn,#imm8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001110 imm3 0 1 0 1</td>
</tr>
</tbody>
</table>

**Encoding T1**

<table>
<thead>
<tr>
<th>ADDS Rd,Rn,#imm3</th>
</tr>
</thead>
<tbody>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
</tr>
</tbody>
</table>

### Instruction Details

- **Encoding T4**
  - ARMv7-M
  - ADDW<IE> Rd,Rn,#imm12
  - If d in \{13,15\} then UNPREDICTABLE;
  - d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
  - If Rn == '1101' then SEE ADD (SP plus immediate);
  - If d in \{13,15\} || n == 15 then UNPREDICTABLE;
  - d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);
  - It can optionally update the condition flags based on the result.

- **Encoding T3**
  - ARMv7-M
  - ADDc{S}W Rd,Rn,#const
  - 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  - 11110i01000S 0 1 1 1 0
  - imm8 Rn imm3 Rd

- **Encoding T2**
  - ARMv7-M
  - ADDS Rd,Rn,#imm8
  - 0001110 imm3 0 1 0 1
  - Rd Rn

- **Encoding T1**
  - ARMv7-M
  - ADDS Rd,Rn,#imm3
  - 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
  - 0 0 0 1 1 1 0 imm3 Rn Rd

### ARMv7-M Architecture Reference Manual
### Table A4-1 Branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>B</em> on page A6-40</td>
<td>Branch to target address</td>
<td>+/-1 MB</td>
</tr>
<tr>
<td><strong>CBNZ, CBZ</strong> on page A6-52</td>
<td>Compare and Branch on Nonzero,</td>
<td>0-126 B</td>
</tr>
<tr>
<td></td>
<td>Compare and Branch on Zero</td>
<td></td>
</tr>
<tr>
<td><em>BL</em> on page A6-49</td>
<td>Call a subroutine</td>
<td>+/-16 MB</td>
</tr>
<tr>
<td><em>BLX (register)</em> on page A6-50</td>
<td>Call a subroutine, optionally change</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>instruction set</td>
<td></td>
</tr>
<tr>
<td><em>BX</em> on page A6-51</td>
<td>Branch to target address, change</td>
<td>Any</td>
</tr>
<tr>
<td></td>
<td>instruction set</td>
<td></td>
</tr>
<tr>
<td><em>TBB, TBH</em> on page A6-258</td>
<td>Table Branch (byte offsets)</td>
<td>0-510 B</td>
</tr>
<tr>
<td></td>
<td>Table Branch (halfword offsets)</td>
<td>0-131070 B</td>
</tr>
</tbody>
</table>
### Table A4-2 Standard data-processing instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.</td>
</tr>
<tr>
<td>ADR</td>
<td>Form PC-relative Address</td>
<td>First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>Bitwise Bit Clear</td>
<td></td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td>Sets flags. Like ADD but with no destination register.</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>Sets flags. Like SUB but with no destination register.</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise Exclusive OR</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Copies operand to destination</td>
<td>Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See <em>Shift instructions</em> on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.</td>
</tr>
</tbody>
</table>

**Many, Many More!**
## Load/Store instructions

Table A4-10 Load and store instructions

<table>
<thead>
<tr>
<th>Data type</th>
<th>Load</th>
<th>Store</th>
<th>Load unprivileged</th>
<th>Store unprivileged</th>
<th>Load exclusive</th>
<th>Store exclusive</th>
</tr>
</thead>
<tbody>
<tr>
<td>32-bit word</td>
<td>LDR</td>
<td>STR</td>
<td>LDRT</td>
<td>STRT</td>
<td>LDREX</td>
<td>STREX</td>
</tr>
<tr>
<td>16-bit halfword</td>
<td>-</td>
<td>STRH</td>
<td>-</td>
<td>STRHT</td>
<td>-</td>
<td>STREXH</td>
</tr>
<tr>
<td>16-bit unsigned halfword</td>
<td>LDRH</td>
<td>-</td>
<td>LDRHT</td>
<td>-</td>
<td>LDREXH</td>
<td>-</td>
</tr>
<tr>
<td>16-bit signed halfword</td>
<td>LDRSH</td>
<td>-</td>
<td>LDRSHT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8-bit byte</td>
<td>-</td>
<td>STRB</td>
<td>-</td>
<td>STRBT</td>
<td>-</td>
<td>STREXB</td>
</tr>
<tr>
<td>8-bit unsigned byte</td>
<td>LDRB</td>
<td>-</td>
<td>LDRBT</td>
<td>-</td>
<td>LDREXB</td>
<td>-</td>
</tr>
<tr>
<td>8-bit signed byte</td>
<td>LDRSB</td>
<td>-</td>
<td>LDRSBT</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>two 32-bit words</td>
<td>LDRD</td>
<td>STRD</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
## Miscellaneous instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Exclusive</td>
<td>CLREX on page A6-56</td>
</tr>
<tr>
<td>Debug hint</td>
<td>DBG on page A6-67</td>
</tr>
<tr>
<td>Data Memory Barrier</td>
<td>DMB on page A6-68</td>
</tr>
<tr>
<td>Data Synchronization Barrier</td>
<td>DSB on page A6-70</td>
</tr>
<tr>
<td>Instruction Synchronization Barrier</td>
<td>ISB on page A6-76</td>
</tr>
<tr>
<td>If Then (makes following instructions conditional)</td>
<td>IT on page A6-78</td>
</tr>
<tr>
<td>No Operation</td>
<td>NOP on page A6-167</td>
</tr>
<tr>
<td>Preload Data</td>
<td>PLD, PLDW (immediate) on page A6-176</td>
</tr>
<tr>
<td></td>
<td>PLD (register) on page A6-180</td>
</tr>
<tr>
<td>Preload Instruction</td>
<td>PLI (immediate, literal) on page A6-182</td>
</tr>
<tr>
<td></td>
<td>PLI (register) on page A6-184</td>
</tr>
<tr>
<td>Send Event</td>
<td>SEV on page A6-212</td>
</tr>
<tr>
<td>Supervisor Call</td>
<td>SVC (formerly SWI) on page A6-252</td>
</tr>
<tr>
<td>Wait for Event</td>
<td>WFE on page A6-276</td>
</tr>
<tr>
<td>Wait for Interrupt</td>
<td>WFI on page A6-277</td>
</tr>
<tr>
<td>Yield</td>
<td>YIELD on page A6-278</td>
</tr>
</tbody>
</table>
### A5.3.2 Modified immediate constants in Thumb instructions

Table A5-11 shows the range of modified immediate constants available in Thumb data processing instructions, and how they are encoded in the a, b, c, d, e, f, g, h, i, and imm3 fields in the instruction.

**Table A5-11 Encoding of modified immediates in Thumb data-processing instructions**

<table>
<thead>
<tr>
<th>i:imm3:a</th>
<th>&lt;const&gt; (^a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000x</td>
<td>00000000 00000000 00000000 abcdefgh</td>
</tr>
<tr>
<td>0001x</td>
<td>00000000 abcdefgh 00000000 abcdefgh (^b)</td>
</tr>
<tr>
<td>0010x</td>
<td>abcdefgh 00000000 abcdefgh 00000000 (^b)</td>
</tr>
<tr>
<td>0011x</td>
<td>abcdefgh abcdefgh abcdefgh abcdefgh (^b)</td>
</tr>
<tr>
<td>01000</td>
<td>1bcdefgh 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01001</td>
<td>01bcdefg h0000000 00000000 00000000</td>
</tr>
<tr>
<td>01010</td>
<td>001bcdef gh000000 00000000 00000000</td>
</tr>
<tr>
<td>01011</td>
<td>0001bcde fgh0000 00000000 00000000</td>
</tr>
<tr>
<td>11101</td>
<td>00000000 00000000 000001bc defgh000</td>
</tr>
<tr>
<td>11110</td>
<td>00000000 00000000 0000001b cdefgh000</td>
</tr>
<tr>
<td>11111</td>
<td>00000000 00000000 00000001 bcdefgh0</td>
</tr>
</tbody>
</table>

---

\(^a\) In this table, the immediate constant value is shown in binary form, to relate abcdefgh to the encoding diagram. In assembly syntax, the immediate value is specified in the usual way (a decimal number by default).

\(^b\) UNPREDICTABLE if abcdefgh == 00000000.
A2.3.2 The Application Program Status Register (APSR)

Program status is reported in the 32-bit Application Program Status Register (APSR), where the defined bits break down into a set of flags as follows:

APSR bit fields are in the following two categories:

- **Reserved bits** are allocated to system features or are available for future expansion. Further information on currently allocated reserved bits is available in *The special-purpose program status registers (xPSR)* on page B1-8. Application level software must ignore values read from reserved bits, and preserve their value on a write. The bits are defined as UNK/SBZP.

- **Flags** that can be set by many instructions:
  
  **N, bit [31]** Negative condition code flag. Set to bit [31] of the result of the instruction. If the result is regarded as a two's complement signed integer, then N == 1 if the result is negative and N = 0 if it is positive or zero.

  **Z, bit [30]** Zero condition code flag. Set to 1 if the result of the instruction is zero, and to 0 otherwise. A result of zero often indicates an equal result from a comparison.

  **C, bit [29]** Carry condition code flag. Set to 1 if the instruction results in a carry condition, for example an unsigned overflow on an addition.

  **V, bit [28]** Overflow condition code flag. Set to 1 if the instruction results in an overflow condition, for example a signed overflow on an addition.

  **Q, bit [27]** Set to 1 if an SSAT or USAT instruction changes (saturates) the input value for the signed or unsigned range of the result.

A2.3.3 Execution state support

ARMv7-M only executes Thumb instructions, and therefore always executes instructions in Thumb state. See Chapter A6 *Thumb Instruction Details* for a list of the instructions supported.

In addition to normal program execution, there is a Debug state – see Chapter C1 *ARMv7-M Debug* for more details.

A2.3.4 Privileged execution

Good system design practice requires the application developer to have a degree of knowledge of the underlying system architecture and the services it offers. System support requires a level of access generally referred to as privileged operation. The system support code determines whether applications run in a privileged or unprivileged manner. Where both privileged and unprivileged support is provided by an operating system, applications usually run unprivileged, allowing the operating system to allocate system resources for sole or shared use by the application, and to provide a degree of protection with respect to other processes and tasks.
Updating the APSR

- **SUB Rx, Ry**
  - \( Rx = Rx - Ry \)
  - APSR unchanged

- **SUBS**
  - \( Rx = Rx - Ry \)
  - APSR N, Z, C, V updated

- **ADD Rx, Ry**
  - \( Rx = Rx + Ry \)
  - APSR unchanged

- **ADD S**
  - \( Rx = Rx + Ry \)
  - APSR N, Z, C, V updated
Conditional execution:
Append to many instructions for conditional execution

### Table A6-1 Condition codes

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point) ab</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS c</td>
<td>Carry set</td>
<td>Greater than, equal, or unordered</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC d</td>
<td>Carry clear</td>
<td>Less than</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus, negative</td>
<td>Less than</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus, positive or zero</td>
<td>Greater than, equal, or unordered</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
<td>C == 1 and Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
<td>C == 0 or Z == 1</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
<td>N ! = V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
<td>Z == 0 and N == V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than, equal, or unordered</td>
<td>Z == 1 or N != V</td>
</tr>
<tr>
<td>1110</td>
<td>None (AL) e</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
<td>Any</td>
</tr>
</tbody>
</table>

a. Unordered means at least one NaN operand.
b. ARMv7-M does not currently support floating point instructions. This column can be ignored.
c. HS (unsigned higher or same) is a synonym for CS.
d. LO (unsigned lower) is a synonym for CC.
e. AL is an optional mnemonic extension for always, except in IT instructions. See IT on page A6-78 for details.
The ARM architecture “books” for this class
The ARM software tools “books” for this class

Sourcery G++ Lite
ARM EABI
Sourcery G++ Lite 2010q1-188
Getting Started

Using as

The GNU Compiler Collection

Using the GNU Compiler Collection

The GNU linker

The GNU Binary Utilities

Debugging with GDB

Dean Eleazer, Jon Freedman & Friends

Richard M. Stallman and the GNU Developer Community

Steve Chambers|tin Jay Leane Taylor

Roland H. Pesch

Cypress Support

Richard Stallman, Roland Pesch, Sean Sibley, et al.
Exercise: What is the value of r2 at \texttt{done}?\

...\
start:\n    \texttt{movs r0, #1}\
    \texttt{movs r1, #1}\
    \texttt{movs r2, #1}\
    \texttt{sub r0, r1}\
    \texttt{bne done}\
    \texttt{movs r2, #2}\
done:\n    \texttt{b done}\
...
Solution: what is the value of r2 at done?

...  

\textbf{start:}

\begin{verbatim}
  \textbf{movs r0, #1} // r0 \leftarrow 1, Z=0 \\
  \textbf{movs r1, #1} // r1 \leftarrow 1, Z=0 \\
  \textbf{movs r2, #1} // r2 \leftarrow 1, Z=0 \\
  \textbf{sub r0, r1} // r0 \leftarrow r0-r1 \\
\end{verbatim}

\begin{verbatim}
  \textbf{subn e done} // NE true when Z==0 \\
\end{verbatim}

\begin{verbatim}
  \textbf{movs r2, #2} // not executed \\
\end{verbatim}

\textbf{done:}

\begin{verbatim}
  \textbf{b done} // r2 is still 1 \\
\end{verbatim}

...
.equ STACK_TOP, 0x20000800
.text
.syntax unified
.thumb
.global _start
.type start, %function

_start:
    .word STACK_TOP, start

start:
    movs r0, #10
   movs r1, #0

loop:
    adds r1, r0
   subs r0, #1
    bne loop

deadloop:
    b deadloop

.end
A simple Makefile

all:
    arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
    arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
    arm-none-eabi-objcopy -Obinary example1.out example.bin
    arm-none-eabi-objdump -S example1.out > example1.list
An ARM assembly language program for GNU

```
.equ STACK_TOP, 0x20000800
.text
.syntax unified
.thumb
.global _start
.type start, %function

_start:
    .word STACK_TOP, start

start:
    movs r0, #10
    movs r1, #0

loop:
    adds r1, r0
    subs r0, #1
    bne loop

deadloop:
    b deadloop
.end
```
Disassembled object code

example1.out: file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
  0: 20000800 .word 0x20000800
  4: 00000009 .word 0x00000009

00000008 <start>:
  8: 200a movs r0, #10
  a: 2100 movs r1, #0

0000000c <loop>:
  c: 1809 adds r1, r1, r0
  e: 3801 subs r0, #1
  10: d1fc bne.n c <loop>

00000012 <deadloop>:
  12: e7fe b.n 12 <deadloop>
Outline

- Minute quiz
- Announcements
- ARM Cortex-M3 ISA
- Assembly tool flow
- C/Assembly mixed tool flow
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)
**Major elements of an Instruction Set Architecture**
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
<td>R4</td>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
<td>R7</td>
<td>R8</td>
<td>R9</td>
<td>R10</td>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
<td>R13 (SP)</td>
<td>R14 (LR)</td>
<td>R15 (PC)</td>
<td>xPSR</td>
<td></td>
</tr>
</tbody>
</table>
**Major elements of an Instruction Set Architecture**

(registers, memory, word size, endianess, conditions, instructions, addressing modes)

<table>
<thead>
<tr>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
</tr>
<tr>
<td>R1</td>
</tr>
<tr>
<td>R2</td>
</tr>
<tr>
<td>R3</td>
</tr>
<tr>
<td>R4</td>
</tr>
<tr>
<td>R5</td>
</tr>
<tr>
<td>R6</td>
</tr>
<tr>
<td>R7</td>
</tr>
<tr>
<td>R8</td>
</tr>
<tr>
<td>R9</td>
</tr>
<tr>
<td>R10</td>
</tr>
<tr>
<td>R11</td>
</tr>
<tr>
<td>R12</td>
</tr>
<tr>
<td><strong>R13 (SP)</strong></td>
</tr>
<tr>
<td>R14 (LR)</td>
</tr>
<tr>
<td>R15 (PC)</td>
</tr>
<tr>
<td>xPSR</td>
</tr>
</tbody>
</table>

### Memory Map

<table>
<thead>
<tr>
<th>Type</th>
<th>Base Address</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>System</td>
<td>0xFFFFFFFF</td>
<td>1 GB</td>
</tr>
<tr>
<td>Private peripheral bus - External</td>
<td>0xE0100000</td>
<td>1 GB</td>
</tr>
<tr>
<td>Private peripheral bus - Internal</td>
<td>0xE0040000</td>
<td>1 GB</td>
</tr>
<tr>
<td>External device</td>
<td>0xA0000000</td>
<td>1 GB</td>
</tr>
<tr>
<td>External RAM</td>
<td>0x60000000</td>
<td>0.5 GB</td>
</tr>
<tr>
<td>Peripheral</td>
<td>0x40000000</td>
<td>0.5 GB</td>
</tr>
<tr>
<td>SRAM</td>
<td>0x20000000</td>
<td>0.5 GB</td>
</tr>
<tr>
<td>Code</td>
<td>0x00000000</td>
<td>0.5 GB</td>
</tr>
</tbody>
</table>
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>32-bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td></td>
</tr>
<tr>
<td>R3</td>
<td></td>
</tr>
<tr>
<td>R4</td>
<td></td>
</tr>
<tr>
<td>R5</td>
<td></td>
</tr>
<tr>
<td>R6</td>
<td></td>
</tr>
<tr>
<td>R7</td>
<td></td>
</tr>
<tr>
<td>R8</td>
<td></td>
</tr>
<tr>
<td>R9</td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td></td>
</tr>
<tr>
<td>R11</td>
<td></td>
</tr>
<tr>
<td><strong>R13 (SP)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>R14 (LR)</strong></td>
<td></td>
</tr>
<tr>
<td><strong>R15 (PC)</strong></td>
<td></td>
</tr>
<tr>
<td>xPSR</td>
<td></td>
</tr>
</tbody>
</table>

32-bits

- System
- Private peripheral bus - External
- Private peripheral bus - Internal
- External device: 1.0 GB
- External RAM: 1.0 GB
- Peripheral: 0.5 GB
- SRAM: 0.5 GB
- Code: 0.5 GB
- Code: 0x00000000
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianness, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R13 (SP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R14 (LR)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>R15 (PC)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>xPSR</td>
</tr>
</tbody>
</table>

32-bits

Endianness

Endianness
### Major elements of an Instruction Set Architecture

(registers, memory, word size, endianess, conditions, instructions, addressing modes)

#### 32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
<th>R13 (SP)</th>
<th>R14 (LR)</th>
<th>R15 (PC)</th>
<th>xPSR</th>
</tr>
</thead>
</table>

#### Endianness

- 32-bits

#### 32-bits

- System
  - 0xFFFFFFF
- Private peripheral bus - External
  - 0xE010000
- Private peripheral bus - Internal
  - 0xE004000
- External device
  - 1.0GB
- External RAM
  - 1.0GB
- Peripheral
  - 0.5GB
- SRAM
  - 0.5GB
- Code
  - 0.5GB
- 0x00000000

#### Endianness
Major elements of an Instruction Set Architecture (registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)
xPSR

Endianness

32-bits

Endianness
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

- **32-bits**
  - R0
  - R1
  - R2
  - R3
  - R4
  - R5
  - R6
  - R7
  - R8
  - R9
  - R10
  - R11
  - R12
  - R13 (SP)
  - R14 (LR)
  - R15 (PC)
  - xPSR

- **32-bits**

```assembly
mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1
```

Endianness

- 32-bits

Endianness

- 32-bits

- System
- Private peripheral bus - External
- Private peripheral bus - Internal
- External device 1.0GB
- External RAM 1.0GB
- Peripheral 0.5GB
- SRAM 0.5GB
- Code 0.5GB
- RESERVE 0x00000000
- 0x00000000

- RESERVE 0x00000000
- 0x00000000
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

### 32-bits

- R0
- R1
- R2
- R3
- R4
- R5
- R6
- R7
- R8
- R9
- R10
- R11
- R12
- R13 (SP)
- R14 (LR)
- R15 (PC)

### 32-bits

- **mov r0, #1**
- **ld r1, [r0,#5]**
  - mem((r0)+5)
- **bne loop**
- **subs r2, #1**

### Endianness

- System
  - 0xFFFFFFF
- Private peripheral bus - External
  - 0xE0100000
- Private peripheral bus - Internal
  - 0xE0040000
- External device
  - 1.0GB
- External RAM
  - 1.0GB
- Peripheral
  - 0.5GB
- SRAM
  - 0.5GB
- Code
  - 0.5GB

### Endianness
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianness, conditions, instructions, addressing modes)

32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
\textbf{R13 (SP)}
\textbf{R14 (LR)}
\textbf{R15 (PC)}

\textbf{xPSR}

Endianness

\begin{align*}
\text{mov} & \ r0, \ #1 \\
\text{ld} & \ r1, \ [r0,#5] \\
\text{mem} & \ (r0)+5 \\
\text{bne} & \ \text{loop} \\
\text{subs} & \ r2, \ #1
\end{align*}

Endianness

32-bits
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
<th>R13 (SP)</th>
<th>R14 (LR)</th>
<th>R15 (PC)</th>
<th>xPSR</th>
</tr>
</thead>
</table>

32-bits

- **mov r0, #1**
- **ld r1, [r0,#5]**
  
  `mem((r0)+5)`

- **bne loop**
- **subs r2, #1**

Endianness

- System
  - 0xFFFFFFFF
  - 0xE0100000
  - 0xE0040000
  - 0xE0000000

- Private peripheral bus - External
  - 0xE0000000

- Private peripheral bus - Internal
  - 0xA0000000

- External device
  - 1.0GB

- External RAM
  - 1.0GB

- Peripheral
  - 0.5GB

- SRAM
  - 0.5GB

- Code
  - 0.5GB

- 0x00000000

RESERVED

N Z C V Q

34
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13 (SP)
R14 (LR)
R15 (PC)

Endianness

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianness
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianness, conditions, instructions, addressing modes)

32-bits

Endianness

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
<th>R10</th>
<th>R11</th>
<th>R12</th>
<th>R13 (SP)</th>
<th>R14 (LR)</th>
<th>R15 (PC)</th>
</tr>
</thead>
</table>

Endianness

```
mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
```

```
sub s r2, #1
```

```
bne loop
```
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianness, conditions, instructions, addressing modes)

32-bits

<table>
<thead>
<tr>
<th>R0</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>R5</th>
<th>R6</th>
<th>R7</th>
<th>R8</th>
<th>R9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R10</td>
<td>R11</td>
<td>R12</td>
<td>R13 (SP)</td>
<td>R14 (LR)</td>
<td>R15 (PC)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>xPSR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Endianness

mov r0, #1
ld r1, [r0,#5]
mem((r0)+5)
bne loop
subs r2, #1

Endianness
# Branch

## Table A4-1 Branch instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Usage</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>B on page A6-40</td>
<td>Branch to target address</td>
<td>+/-1 MB</td>
</tr>
<tr>
<td>CBNZ, CBZ on page A6-52</td>
<td>Compare and Branch on Nonzero, Compare and Branch on Zero</td>
<td>0-126 B</td>
</tr>
<tr>
<td>BL on page A6-49</td>
<td>Call a subroutine</td>
<td>+/-16 MB</td>
</tr>
<tr>
<td>BLX (register) on page A6-50</td>
<td>Call a subroutine, optionally change instruction set</td>
<td>Any</td>
</tr>
<tr>
<td>BX on page A6-51</td>
<td>Branch to target address, change instruction set</td>
<td>Any</td>
</tr>
<tr>
<td>TBB, TBH on page A6-258</td>
<td>Table Branch (byte offsets)</td>
<td>0-510 B</td>
</tr>
<tr>
<td></td>
<td>Table Branch (halfword offsets)</td>
<td>0-131070 B</td>
</tr>
</tbody>
</table>
## Branch Encoding

### Encoding T1

All versions of the Thumb instruction set.  
\[ B<c> \ <\text{label}> \]

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 0 1 | \( \text{cond} \) | \( \text{imm8} \) |

if \( \text{cond} = '1110' \) then UNDEFINED;  
if \( \text{cond} = '1111' \) then SEE SVC;  
\( \text{imm32} = \text{SignExtend}(\text{imm8}:'0', 32) \);  
if InITBlock() then UNPREDICTABLE;

### Encoding T2

All versions of the Thumb instruction set.  
\[ B<c> \ <\text{label}> \]

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 0 0 | \( \text{imm11} \) |

\( \text{imm32} = \text{SignExtend}(\text{imm11}:'0', 32) \);

if InITBlock() \&\& !LastInITBlock() then UNPREDICTABLE;

### Encoding T3

ARMv7-M  
\[ B<c> .W \ <\text{label}> \]

Not allowed in IT block.

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 | \( \text{S} \) | \( \text{cond} \) | \( \text{imm6} \) |
| 0 | \( \text{J1} \) | \( \text{J2} \) |

### Encoding T4

ARMv7-M  
\[ B<c> .W \ <\text{label}> \]

Outside or last in IT block

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 1 1 1 0 | \( \text{S} \) | \( \text{imm10} \) | \( \text{J1} \) | \( \text{J2} \) |
| \( \text{imm11} \) |

\( I1 = \text{NOT}(J1 \text{ EOR } S) \);  
\( I2 = \text{NOT}(J2 \text{ EOR } S) \);  
\( \text{imm32} = \text{SignExtend}(\text{S}:I1:I2:imm10:imm11:'0', 32) \);  
if InITBlock() \&\& !LastInITBlock() then UNPREDICTABLE;
## Conditional execution:
Append to many instructions for conditional execution

### Table A6-1 Condition codes

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point)</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS&lt;sup&gt;c&lt;/sup&gt;</td>
<td>Carry set</td>
<td>Greater than, equal, or unordered</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC&lt;sup&gt;d&lt;/sup&gt;</td>
<td>Carry clear</td>
<td>Less than</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus, negative</td>
<td>Less than</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus, positive or zero</td>
<td>Greater than, equal, or unordered</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
<td>C == 1 and Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
<td>C == 0 or Z == 1</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
<td>N != V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
<td>Z == 0 and N == V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than, equal, or unordered</td>
<td>Z == 1 or N != V</td>
</tr>
<tr>
<td>1110</td>
<td>None (AL)&lt;sup&gt;e&lt;/sup&gt;</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
<td>Any</td>
</tr>
</tbody>
</table>

---

a. Unordered means at least one NaN operand.
b. ARMv7-M does not currently support floating point instructions. This column can be ignored.
c. HS (unsigned higher or same) is a synonym for CS.
d. LO (unsigned lower) is a synonym for CC.
e. AL is an optional mnemonic extension for always, except in IT instructions. See IT on page A6-78 for details.
The ITE Instruction

• Useful for small If-Then-Else constructs

• \texttt{IT<x><y><z> <cond>}

  – \( <x>, <y>, <z> \) can be T or E which refers to the base condition \( <\text{cond}> \)

Here is an example of IT use:

```c
if (R0 equal R1) then {
  R3 = R4 + R5
  R3 = R3 / 2
} else {
  R3 = R6 + R7
  R3 = R3 / 2
}
```

This can be written as:

```c
CMP   R0, R1     ; Compare R0 and R1
ITTEE EQ         ; If R0 equal R1, Then-Then-Else-Else
ADDEQ R3, R4, R5 ; Add if equal
ASREQ R3, R3, #1 ; Arithmetic shift right if equal
ADDNE R3, R6, R7 ; Add if not equal
ASRNE R3, R3, #1 ; Arithmetic shift right if not equal
```

CBZ and CBNZ

The compare and then branch if zero/nonzero instructions are useful for looping (for example, the WHILE loop in C). The syntax is:

```c
CBZ  <Rn>, <label>
```

or:

```c
CBNZ <Rn>, <label>
```

where \( <\text{label}> \) is a forward branch address. For example:

```c
while (R0 != 0) {
  function1();
}
```

This can be written as:

```c
... loop
CBZ R0, loopexit
BL  function1
B   loop
loopexit
...
```

Flags are not affected by this instruction.
Instruction classes
Instruction classes

• Branching
Instruction classes

- Branching
Instruction classes

• Branching

• Data processing
Instruction classes

- Branching
- Data processing
Instruction classes

- Branching
- Data processing
- Load/store
Instruction classes

- Branching
- Data processing
- Load/store
Instruction classes

- Branching
- Data processing
- Load/store
- Exceptions
Instruction classes

- Branching
- Data processing
- Load/store
- Exceptions
Instruction classes

• Branching
• Data processing
• Load/store
• Exceptions
• Miscellaneous
Updating the Application Program Status Register (aka condition codes or APRS)

- **sub r0, r1**
  - \( r0 \leftarrow r0 - r1 \)
  - APSR remain unchanged

- **subs r0, r1**
  - \( r0 \leftarrow r0 - r1 \)
  - APSR N or Z bits could change

- **add r0, r1**
  - \( r0 \leftarrow r0 + r1 \)
  - APSR remain unchanged

- **adds r0, r1**
  - \( r0 \leftarrow r0 + r1 \)
  - APSR C or V bits could change
What does some real assembly look like?

```assembly
0000017c <main>:
    b580      push  {r7, lr}
    b084      sub   sp, #16
    af00      add   r7, sp, #0
    f04f 0328  mov.w r3, #40 ; 0x28
    60bb      str   r3, [r7, #8]
    f04f 0300  mov.w r3, #0
    60fb      str   r3, [r7, #12]
    f04f 0300  mov.w r3, #0
    603b      str   r3, [r7, #0]
    e010      b.n   1b8 <main+0x3c>
    6838      ldr   r0, [r7, #0]
    f7ff ffb8  bl    10c <factorial>
    4603      mov   r3, r0
    607b      str   r3, [r7, #4]
    f646 5010  movw  r0, #27920 ; 0x6d10
    f2c0 0000  movt  r0, #0
    6839      ldr   r1, [r7, #0]
    687a      ldr   r2, [r7, #4]
    f000 f840  bl    230 <printf>
    683b      ldr   r3, [r7, #0]
...
Syntax

• .N
  – Meaning narrow, specifies that the assembler must select a 16-bit encoding for the instruction. If this is not possible, an assembler error is produced.

• .W
  – Meaning wide, specifies that the assembler must select a 32-bit encoding for the instruction. If this is not possible, an assembler error is produced.
The endianness religious war: 286 years and counting!

- Modern version
  - Danny Cohen
  - IEEE Computer, v14, #10
  - Published in 1981
  - Satire on CS religious war

- Historical Inspiration
  - Jonathan Swift
  - Gullivers Travels
  - Published in 1726
  - Satire on Henry-VIII’s split with the Church

- Little-Endian
  - LSB is at lower address

<table>
<thead>
<tr>
<th>Memory Offset</th>
<th>Value (LSB) (MSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>01 02 FF 00</td>
</tr>
</tbody>
</table>

```c
uint8_t a = 1;
uint8_t b = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;
```

- Big-Endian
  - MSB is at lower address

<table>
<thead>
<tr>
<th>Memory Offset</th>
<th>Value (LSB) (MSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0004</td>
<td>78 56 34 12</td>
</tr>
</tbody>
</table>

```c
uint8_t a = 1;
uint8_t b = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;
```
### Instruction encoding

- Instructions are encoded in machine language opcodes
- Sometimes
  - Necessary to hand generate opcodes
  - Necessary to verify assembled code is correct

### How?

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Register Value</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>movs r0, #10</td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td></td>
<td>(msb)</td>
<td>0a 20 00 21 (lsb)</td>
</tr>
<tr>
<td>movs r1, #0</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>

**Encoding T1**

All versions of the Thumb instruction set.

**ARMv7 ARM**

- MOVs <Rd>,#<imm8>
- MOV<c> <Rd>,#<imm8>

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-12</td>
<td>Rd</td>
</tr>
<tr>
<td>11-8</td>
<td>imm8</td>
</tr>
</tbody>
</table>
What happens after a power-on-reset (POR)?

- On the ARM Cortex-M3
- SP and PC are loaded from the code (.text) segment
- **Initial stack pointer**
  - LOC: 0x00000000
  - POR: SP ← mem(0x00000000)
- **Interrupt vector table**
  - *Initial* base: 0x00000004
  - Vector table is relocatable
  - Entries: 32-bit values
  - Each entry is an address
  - Entry #1: reset vector
    - LOC: 0x00000004
    - POR: PC ← mem(0x00000004)
- **Execution begins**
Outline

- Minute quiz
- Announcements
- ARM Cortex-M3 ISA
- Assembly tool flow
- C/Assembly mixed tool flow
How does an assembly language program get turned into a executable program image?

Assembly files (.s) → Object files (.o) → Executable image file

- **Assembly files (.s)** are created by an assembler program.
- **Object files (.o)** are created by the assembler program.
- **Executable image file** is created by the linker program.

**Tools**:
- `as` (assembler)
- `ld` (linker)
- `objcopy`
- `objdump`
What are the real GNU executable names for the ARM?

- Just add the prefix “arm-none-eabi-” prefix
- Assembler (as)
  - arm-none-eabi-as
- Linker (ld)
  - arm-none-eabi-ld
- Object copy (objcopy)
  - arm-none-eabi-objcopy
- Object dump (objdump)
  - arm-none-eabi-objdump
- C Compiler (gcc)
  - arm-none-eabi-gcc
- C++ Compiler (g++)
  - arm-none-eabi-g++
A simple (hardcoded) Makefile example

all:
  arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
  arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
  arm-none-eabi-objcopy -Obinary example1.out example1.bin
  arm-none-eabi-objdump -S example1.out > example1.lst
What information does the disassembled file provide?

The disassembled file provides the disassembly of the example1.out file, which is an object file created from the source file example1.s. The file format is elf32-littlearm.

```
.equ STACK_TOP, 0x20000800
.text
.syntax unified
.thumb
.global _start
.type start, %function

_start:
.word STACK_TOP, start

start:
  movs r0, #10
  movs r1, #0

loop:
  adds r1, r0
  subs r0, #1
  bne loop

deadloop:
  b deadloop
.end
```

Disassembly of section .text:

```
00000000 <_start>:
  0: 20000800 .word 0x20000800
  4: 00000009 .word 0x00000009

00000008 <start>:
  8: 200a moves r0, #10
  a: 2100 moves r1, #0

0000000c <loop>:
  c: 1809 adds r1, r1, r0
  e: 3801 subs r0, #1
  10: d1fc bne.n c <loop>

00000012 <deadloop>:
  12: e7fe b.n 12 <deadloop>
```
What are the elements of a real assembly program?

```
.equ    STACK_TOP, 0x20000800  /* Equates symbol to value */
.text   /* Tells AS to assemble region */
.syntax unified /* Means language is ARM UAL */
.thumb  /* Means ARM ISA is Thumb */
.global _start /* .global exposes symbol */
    /* _start label is the beginning */
    /* ...of the program region */
.type   start, %function /* Specifies start is a function */
    /* start label is reset handler */

_start:
    .word  STACK_TOP, start /* Inserts word 0x20000800 */
    /* Inserts word (start) */

start:
    movs r0, #10 /* We’ve seen the rest ... */
    movs r1, #0

loop:
    adds r1, r0
    subs r0, #1
    bne  loop

deadloop:
    b    deadloop
.end
```
.equ STACK_TOP, 0x20000800 /* Equates symbol to value */
.text /* Tells AS to assemble region */
syntax unified /* Means language is ARM UAL */
.thumb /* Means ARM ISA is Thumb */
global _start /* .global exposes symbol */
    /* _start label is the beginning */
    /* ...of the program region */
    type start, %function /* Specifies start is a function */
    /* start label is reset handler */

_start:
.word STACK_TOP, start /* Inserts word 0x20000800 */
    /* Inserts word (start) */

start:
    movs r0, #10
    movs r1, #0
    /* We’ve seen the rest ... */

loop:
    adds r1, r0
    subs r0, #1
    bne loop

deadloop:
    b deadloop
.end
How are assembly files assembled?

- $ arm-none-eabi-as
  - Useful options
    - -mcpu
    - -mthumb
    - -o

$ arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
How can the contents of an object file be read?

- `$ readelf -a example.o`
  - arm-none-eabi-readelf.exe on Windows

- Shows
  - ELF headers
  - Program headers
  - Section headers
  - Symbol table
  - Files attributes

- Other options
  - `-s` shows symbols
  - `-S` shows section headers
What does an object file contain?

$ readelf -S example1.o
There are 9 section headers, starting at offset 0xac:

Section Headers:

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NULL</td>
<td></td>
<td>00000000</td>
<td>00000</td>
<td>00000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.text</td>
<td>PROGBITS</td>
<td>00000000</td>
<td>00034</td>
<td>00014</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>.rel.text</td>
<td>REL</td>
<td>00000000</td>
<td>00300</td>
<td>00008</td>
<td>08</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.data</td>
<td>PROGBITS</td>
<td>00000000</td>
<td>00048</td>
<td>00000</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>.bss</td>
<td>NOBITS</td>
<td>00000000</td>
<td>00048</td>
<td>00000</td>
<td>0</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>.ARM.attributes</td>
<td>ARM_ATTRIBUTES</td>
<td>00000000</td>
<td>00048</td>
<td>00021</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>00069</td>
<td>00040</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000</td>
<td>00214</td>
<td>000c0</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>002d4</td>
<td>0002c</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Key to Flags:

W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
How are object files linked?

- $ arm-none-eabi-ld
  - Useful options
    - -Ttext
    - -Tbss
    - -o

$ arm-none-eabi-ld -Ttext 0x0 -Tbss 0x20000000 -o example1.out example1.o
What are the contents of typical linker script?

```plaintext
OUTPUT_FORMAT("elf32-littlearm", "elf32-bigarm", "elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
  .text :
  {
    . = ALIGN(4);
    *(.text*)
      . = ALIGN(4);
      _etext = .;
  } >ram
}
end = .;
```
What does an executable image file contain?

- **.text segment**
  - Executable code
  - Initial reset vector

- **.data segment (.rodata in ELF)**
  - Static (initialized) variables

- **.bss segment**
  - Static (uninitialized) variables
  - Zero-filled by CRT or OS
  - From: Block Started by Symbol

- Does **not** contain heap or stack

- For details, see: 
  /usr/include/linux/elf.h
How can the contents of an executable file be read?

- Exactly the same way as an object file!

- Recall the useful options
  - -a show all information
  - -s shows symbols
  - -S shows section headers
What does an executable file contain?

- Use readelf’s –S option
- Note that the .out has fewer sections than the .o file
  - Why?

$ readelf -S example1.out
There are 6 section headers, starting at offset 0x8068:

Section Headers:

<table>
<thead>
<tr>
<th>Nr</th>
<th>Name</th>
<th>Type</th>
<th>Addr</th>
<th>Off</th>
<th>Size</th>
<th>ES</th>
<th>Flg</th>
<th>Lk</th>
<th>Inf</th>
<th>Al</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>NULL</td>
<td>00000000</td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>.text</td>
<td>PROGBITS</td>
<td>00000000</td>
<td>0080000</td>
<td>0000140</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.ARM.attributes</td>
<td>ARM_ATTRIBUTES</td>
<td>00000000</td>
<td>0080140</td>
<td>0000210</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>0080350</td>
<td>0000310</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000</td>
<td>0081580</td>
<td>000130</td>
<td>10</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>0082880</td>
<td>0000850</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

Key to Flags:
W (write), A (alloc), X (execute), M (merge), S (strings)
I (info), L (link order), G (group), x (unknown)
O (extra OS processing required) o (OS specific), p (processor specific)
What are the contents of an executable’s .text segment?

- Use readelf’s –x option to hex dump a section
- 1\textsuperscript{st} column shows memory address
- 2\textsuperscript{nd} through 5\textsuperscript{th} columns show data
- The initial SP and PC values are visible
- The executable opcodes are also visible

```
$ readelf -x .text example1.out

Hex dump of section `.text':
  0x00000000 | 00080020 09000000 0a200021 09180138 ... ...... !....8
  0x00000010 | fcd1fee7
```

61
What are the raw contents of an executable file?

- Use hexdump
- ELF’s magic number is visible
- The initial SP, PC, executable opcodes are visible

```
$ hexdump example1.out
0000000 457f 464c 0101 0000 0000 0000 0000 0000
0000010 0002 0028 0001 0000 0000 0000 0000 0034
0000020 8068 0000 0000 0500 0034 0020 0001 0028
0000030 0006 0003 0001 0000 8000 0000 0000 0000
0000040 0000 0000 0014 0000 0014 0000 0005 0000
0000050 8000 0000 0000 0000 0000 0000 0000 0000
0000060 0000 0000 0000 0000 0000 0000 0000 0000
*
0008000 0800 2000 0009 0000 200a 2100 1809 3801
0008010 d1fc e7fe 2041 0000 6100 6165 6962 0100
0008020 0016 0000 4305 524f 4554 2d58 334d 0600
0008030 070a 094d 0002 732e 6d79 6174 0062 732e
0008040 7274 6174 0062 732e 7368 7274 6174 0062
0008050 742e 7865 0074 412e 4d52 612e 7474 6972
0008060 7562 6574 0073 0000 0000 0000 0000 0000
0008070 0000 0000 0000 0000 0000 0000 0000 0000
*
0008090 001b 0000 0001 0000 0006 0000 0000 0000
```
What purpose does an executable file serve?

- Serves as a convenient container for sections/segments
- Keeps segments segregate by type and access rights
- Serves as a program “image” for operating systems
- Allows the loader to place segments into main memory
- Can integrates symbol table and debugging information
How useful is an executable image for most embedded systems & tools?
What does a binary program image contain?

- Basically, a binary copy of program’s `.text` section
- Try ‘hexdump –C example.bin’
- Want to change the program?
  - Try ‘hexedit example.bin’
  - You can change the program (e.g. opcodes, static data, etc.)
- The initial `SP`, `PC`, executable opcodes are visible

```
$ hexdump -C example.bin
00000000 00 08 00 20 09 00 00 00 0a 20 00 21 09 18 01 38 |....|
00000010 fc d1 fe e7 |....|
00000014

$ hexedit example.bin
00000000 00 08 00 20 09 00 00 00 0a 20 00 21 09 18 01 38 ....
00000010 FC D1 FE E7 ....
```
What are other, more usable formats?

- .o, .out, and .bin are all binary formats
- Many embedded tools don’t use binary formats
- Two common ASCII formats
  - Intel hex (ihex)
  - Motorola S-records (srec)
- The initial SP, PC, executable opcodes are visible

```bash
$ arm-none-eabi-objcopy -O ihex example1.out "example1.hex"
$ cat example1.hex
:0000000000000000000000000A200021091801381A
:04001000FCD1FEE73A
:000000001FF

$ arm-none-eabi-objcopy -O srec example1.out "example1.srec"
$ cat example1.srec
S01000006578616D706C65312E73726563F7
S113000000080020900000000A2000210918013816
S1070010FCD1FEE736
S9030000FC
```
• Minute quiz

• Announcements

• ARM Cortex-M3 ISA

• Assembly tool flow

• C/Assembly mixed tool flow
How does an assembly language program get turned into an executable program image?

Assembly files (.s) → Object files (.o) → ld (linker) → Executable image file

- **Assembly files (.s)**
- **Object files (.o)**
- **ld (linker)**
- **Executable image file**
  - **Memory layout**
  - **Linker script (.ld)**

Tools:
- **objcopy**
- **objdump**

Binary program file (.bin)

Disassembled code (.lst)
How does a mixed C/Assembly program get turned into a executable program image?

- **C files (.c)**
- **Assembly files (.s)**
- **Object files (.o)**
- **Library object files (.o)**
- **Memory layout**
- **Linker script (.ld)**
- **ld (linker)**
- **gcc (compile + link)**
- **Executable image file**
- **Binary program file (.bin)**
- **Disassembled code (.lst)**
- **objdump**
- **objcopy**

The process involves compiling C files with `gcc` and assembling Assembly files with `as`. The object files are then combined with `ld`, linking in library object files and using a linker script. The final executable image file can be converted to a binary program file with `objcopy`, and its disassembly can be viewed using `objdump`.
int main() {
    int i;
    int n;
    unsigned int input = 40, output = 0;
    for (i = 0; i < 10; ++i) {
        n = factorial(i);
        printf("factorial(\%d) = \%d\n", i, n);
    }
    __asm("nop\n");
    __asm("mov r0, %0\n"
        "mov r3, #5\n"
        "udiv r0, r0, r3\n"
        "mov %1, r0\n"
        ":=r" (output)
        : "r" (input)
        : "cc", "r3" );
    __asm("nop\n");
    printf("%d\n", output);
}
int main() {
    int i;
    int n;
    unsigned int input = 40, output = 0;
    for (i = 0; i < 10; ++i) {
        n = factorial(i);
        printf("factorial(%d) = %d\n", i, n);
    }
    __asm("nop\n");
    __asm("mov r0, %0\n"
        "mov r3, #5\n"
        "udiv r0, r0, r3\n"
        "mov %1, r0\n"
        ":="r" (output)
        : "r" (input)
        : "cc", "r3" );
    __asm("nop\n");
    printf("%d\n", output);
}

Cheap trick: use asm() or __asm() macros to sprinkle simple assembly in standard C code!

$ arm-none-eabi-gcc \
   -mcpu=cortex-m3 \
   -mthumb main.c \
   -T generic-hosted.ld \
   -o factorial
$ qemu-arm -cpu cortex-m3 \
   ./factorial
factorial(0) = 1
factorial(1) = 1
factorial(2) = 2
factorial(3) = 6
factorial(4) = 24
factorial(5) = 120
factorial(6) = 720
factorial(7) = 5040
factorial(8) = 40320
factorial(9) = 362880
int main() {
    int i;
    int n;
    unsigned int input = 40, output = 0;
    for (i = 0; i < 10; ++i) {
        n = factorial(i);
        printf("factorial(%d) = %d\n", i, n);
    }
    __asm("nop\n");
    __asm("mov r0, %0\n"
           "mov r3, #5\n"
           "udiv r0, r0, r3\n"
           "mov %1, r0\n"
           ":="r" (output)
           : "r" (input)
           : "cc", "r3" );
    __asm("nop\n");
    printf("%d\n", output);
}

Answer: 40/5 → 8
### Table A4-2 Standard data-processing instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Instruction</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADC</td>
<td>Add with Carry</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>Add</td>
<td>Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.</td>
</tr>
<tr>
<td>ADR</td>
<td>Form PC-relative Address</td>
<td>First operand is the PC. Second operand is an immediate constant. Thumb supports a zero-extended 12-bit immediate constant. Operation is an addition or a subtraction.</td>
</tr>
<tr>
<td>AND</td>
<td>Bitwise AND</td>
<td></td>
</tr>
<tr>
<td>BIC</td>
<td>Bitwise Bit Clear</td>
<td></td>
</tr>
<tr>
<td>CMN</td>
<td>Compare Negative</td>
<td>Sets flags. Like ADD but with no destination register.</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare</td>
<td>Sets flags. Like SUB but with no destination register.</td>
</tr>
<tr>
<td>EOR</td>
<td>Bitwise Exclusive OR</td>
<td></td>
</tr>
<tr>
<td>MOV</td>
<td>Copies operand to destination</td>
<td>Has only one operand, with the same options as the second operand in most of these instructions. If the operand is a shifted register, the instruction is an LSL, LSR, ASR, or ROR instruction instead. See <em>Shift instructions</em> on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.</td>
</tr>
</tbody>
</table>

---

**Many, Many More!**

ARmv7-M Architecture Reference Manual
### Table A4-12 Miscellaneous instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clear Exclusive</td>
<td><em>CLREX</em> on page A6-56</td>
</tr>
<tr>
<td>Debug hint</td>
<td><em>DBG</em> on page A6-67</td>
</tr>
<tr>
<td>Data Memory Barrier</td>
<td><em>DMB</em> on page A6-68</td>
</tr>
<tr>
<td>Data Synchronization Barrier</td>
<td><em>DSB</em> on page A6-70</td>
</tr>
<tr>
<td>Instruction Synchronization Barrier</td>
<td><em>ISB</em> on page A6-76</td>
</tr>
<tr>
<td>If Then (makes following instructions conditional)</td>
<td><em>IT</em> on page A6-78</td>
</tr>
<tr>
<td>No Operation</td>
<td><em>NOP</em> on page A6-167</td>
</tr>
<tr>
<td>Preload Data</td>
<td><em>PLD, PLDW (immediate)</em> on page A6-176</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Preload Instruction</td>
<td><em>PLI (immediate, literal)</em> on page A6-182</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Send Event</td>
<td><em>SEV</em> on page A6-212</td>
</tr>
<tr>
<td>Supervisor Call</td>
<td><em>SVC (formerly SWI)</em> on page A6-252</td>
</tr>
<tr>
<td>Wait for Event</td>
<td><em>WFE</em> on page A6-276</td>
</tr>
<tr>
<td>Wait for Interrupt</td>
<td><em>WFI</em> on page A6-277</td>
</tr>
<tr>
<td>Yield</td>
<td><em>YIELD</em> on page A6-278</td>
</tr>
</tbody>
</table>
Conditional execution: Append to many instructions for conditional execution

Table A6-1 Condition codes

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point)</th>
<th>Condition flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>EQ</td>
<td>Equal</td>
<td>Equal</td>
<td>Z == 1</td>
</tr>
<tr>
<td>0001</td>
<td>NE</td>
<td>Not equal</td>
<td>Not equal, or unordered</td>
<td>Z == 0</td>
</tr>
<tr>
<td>0010</td>
<td>CS (^c)</td>
<td>Carry set</td>
<td>Greater than, equal, or unordered</td>
<td>C == 1</td>
</tr>
<tr>
<td>0011</td>
<td>CC (^d)</td>
<td>Carry clear</td>
<td>Less than</td>
<td>C == 0</td>
</tr>
<tr>
<td>0100</td>
<td>MI</td>
<td>Minus, negative</td>
<td>Less than</td>
<td>N == 1</td>
</tr>
<tr>
<td>0101</td>
<td>PL</td>
<td>Plus, positive or zero</td>
<td>Greater than, equal, or unordered</td>
<td>N == 0</td>
</tr>
<tr>
<td>0110</td>
<td>VS</td>
<td>Overflow</td>
<td>Unordered</td>
<td>V == 1</td>
</tr>
<tr>
<td>0111</td>
<td>VC</td>
<td>No overflow</td>
<td>Not unordered</td>
<td>V == 0</td>
</tr>
<tr>
<td>1000</td>
<td>HI</td>
<td>Unsigned higher</td>
<td>Greater than, or unordered</td>
<td>C == 1 and Z == 0</td>
</tr>
<tr>
<td>1001</td>
<td>LS</td>
<td>Unsigned lower or same</td>
<td>Less than or equal</td>
<td>C == 0 or Z == 1</td>
</tr>
<tr>
<td>1010</td>
<td>GE</td>
<td>Signed greater than or equal</td>
<td>Greater than or equal</td>
<td>N == V</td>
</tr>
<tr>
<td>1011</td>
<td>LT</td>
<td>Signed less than</td>
<td>Less than, or unordered</td>
<td>N != V</td>
</tr>
<tr>
<td>1100</td>
<td>GT</td>
<td>Signed greater than</td>
<td>Greater than</td>
<td>Z == 0 and N == V</td>
</tr>
<tr>
<td>1101</td>
<td>LE</td>
<td>Signed less than or equal</td>
<td>Less than, equal, or unordered</td>
<td>Z == 1 or N != V</td>
</tr>
<tr>
<td>1110</td>
<td>None (AL) (^e)</td>
<td>Always (unconditional)</td>
<td>Always (unconditional)</td>
<td>Any</td>
</tr>
</tbody>
</table>

a. Unordered means at least one NaN operand.
b. ARMv7-M does not currently support floating point instructions. This column can be ignored.
c. HS (unsigned higher or same) is a synonym for CS.
d. LO (unsigned lower) is a synonym for CC.
e. AL is an optional mnemonic extension for always, except in IT instructions. See IT on page A6-78 for details.