EECS 373
Introduction to Embedded System Design

Robert Dick
University of Michigan

Lecture 2: Architecture, Assembly, and ABI

16 and 18 January 2024
Review

- Course staff
- Implementation technology trends
- Application trends
- Course structure and grading
- Introduction to debugging
Outline

- ISA
- ABI
- Build process
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

mov r0, #0x1
ld r1, [r0,#5]
r1=mem((r0)+5)
bne loop
subs r2, #1
**Endianness**

- **Little-Endian (default)**
  - LSB(yte) is at lower address

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

- **Big-Endian**
  - MSB(yte) is at lower address

<table>
<thead>
<tr>
<th>Memory Offset (LSB) (MSB)</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0000</td>
<td>01 02 FF 00</td>
</tr>
<tr>
<td>0x0004</td>
<td>12 34 56 78</td>
</tr>
</tbody>
</table>

```c
uint8_t a = 1;
uint8_t b = 2;
uint16_t c = 255; // 0x00FF
uint32_t d = 0x12345678;
```
Endianness: ordering of bytes within a word

- Little - increasing numeric significance with increasing memory addresses
- Big – the opposite, most significant byte first
- MIPS is big endian, x86 is little endian, ARM supports both (generally little)
Instruction encoding

- Instructions are encoded in machine language opcodes

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Register Value</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>movs r0, #10</code></td>
<td>001</td>
<td>00</td>
</tr>
<tr>
<td><code>movs r1, #0</code></td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>

ARMv7

Encodings: T1

MOV $<rd>, $<imm8>

MOV $<c>, $<imm8>

Outside IT block.

Inside IT block.

d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = APSR.C;
Assembly example

data:
    .byte 0x12, 20, 0x20, -1
func:
    mov r0, #0
    mov r4, #0
    movw r1, #:lower16:data
    movt r1, #:upper16:data
top:
    ldrb r2, [r1],#1
    add r1, r1, #1
    add r4, r4, r2
    add r0, r0, #1
    cmp r0, #4
    bne top
Instructions used

• **mov**
  • Moves data from register or immediate.
  • Or also from shifted register or immediate!
    • the mov assembly instruction maps to a bunch of different encodings!
  • If immediate it might be a 16-bit or 32-bit instruction.
    • Not all values possible
      • why?

• **movw**
  • Actually an alias to mov.
    • “w” is “wide”
      • hints at 16-bit immediate.
There are similar entries for move immediate, move shifted (which actually maps to different instructions), etc.
Directives

• #:lower16:data
  • What does that do?
  • Why?
Move Top writes an immediate value to the top halfword of the destination register. It does not affect the contents of the bottom halfword.

**Encoding T1** ARMv7-M

\[
\begin{array}{cccccccccccccccccccc}
15 & 14 & 13 & 12 & 11 & 10 & 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & \text{imm4} & 0 & \text{imm3} & \text{Rd} & \text{imm8} \\
1 & 1 & 1 & 1 & 0 & i & 1 & 0 & 1 & 1 & 0 & 0 & & & & & & & & \\
\end{array}
\]

\( d = \text{UInt(Rd)}; \ \text{imm16} = \text{imm4}:i:\text{imm3}:\text{imm8}; \)
\( \text{if } d \ \text{IN} \ \{13,15\} \ \text{then UNPREDICTABLE}; \)

**Assembler syntax**

\[
\text{MOVT}<c><q> \ <Rd>, #<imm16>
\]

where:

\(<c><q> \quad \text{See Standard assembler syntax fields on page A6-7.}\)

\(<Rd> \quad \text{Specifies the destination register.}\)

\(<\text{imm16}> \quad \text{Specifies the immediate value to be written to} <Rd>. \text{It must be in the range 0-65535.}\)

**Operation**

\(\text{if ConditionPassed() then}\)
\(\quad \text{EncodingSpecificOperations();}\)
\(\quad \text{R}[d]<31:16> = \text{imm16};\)
\(\quad // \ R[d]<15:0> \ \text{unchanged}\)

**Exceptions**

None.
Loads

- **ldrb** -- Load register byte
  - Note this takes an 8-bit value and moves it into a 32-bit location!
  - Zeros out the top 24 bits.

- **ldrsb** -- Load register signed byte
  - Note this also takes an 8-bit value and moves it into a 32-bit location!
  - Uses sign extension for the top 24 bits.
  - Why? Two’s complement makes math efficient.
  - \[-1 \rightarrow 00000001 \rightarrow 11111110 \rightarrow 11111111\]
  - Would otherwise become 255.
Addressing modes

- **Offset addressing**
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - \([<Rn>, \text{<offset>}]\)
- **Pre-indexed addressing**
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - \([<Rn>, \text{<offset>}]\)
- **Post-indexed addressing**
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - \([<Rn>], \text{<offset>}\)
What does the program _do_?

data:
   .byte 0x12, 20, 0x20, -1
func:
   mov r0, #0
   mov r4, #0
   movw r1, #:lower16:data
   movt r1, #:upper16:data
   top:
      ldrb r2, [r1],#1
      add r4, r4, r2
      add r0, r0, #1
      cmp r0, #4
      bne top
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
  - 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
How to read the ARM ARM

- Skim pages 1-84.
- Read pages 85-154.
- Refer to pages 154-end.
A quick comment on the ISA:

A4.1 About the instruction set

ARMv7-M supports a large number of 32-bit instructions that were introduced as Thumb-2 technology into the Thumb instruction set. Much of the functionality available is identical to the ARM instruction set supported alongside the Thumb instruction set in ARMv6T2 and other ARMv7 profiles. This chapter describes the functionality available in the ARMv7-M Thumb instruction set, and the Unified Assembler Language (UAL) that can be assembled to either the Thumb or ARM instruction sets.

Thumb instructions are either 16-bit or 32-bit, and are aligned on a two-byte boundary. 16-bit and 32-bit instructions can be intermixed freely. Many common operations are most efficiently executed using 16-bit instructions. However:

- Most 16-bit instructions can only access eight of the general purpose registers, R0-R7. These are known as the low registers. A small number of 16-bit instructions can access the high registers, R8-R15.

- Many operations that would require two or more 16-bit instructions can be more efficiently executed with a single 32-bit instruction.

The ARM and Thumb instruction sets are designed to interwork freely. Because ARMv7-M only supports Thumb instructions, interworking instructions in ARMv7-M must only reference Thumb state execution, see ARMv7-M and interworking support for more details.

In addition, see:
- Chapter A5 Thumb Instruction Set Encoding for encoding details of the Thumb instruction set
- Chapter A6 Thumb Instruction Details for detailed descriptions of the instructions.
ARM Cortex-M3 ISA

Instruction Set

ADD Rd, Rn, <op2>

Branching
Data processing
Load/Store
Exceptions
Miscellaneous

Register Set

32-bits

Register Set

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

Address Space

32-bits

32-bits

Endianess

Endianess
Registers

Mode dependent
Address space

- ROM Table
- External PPB
- ETM
- TPIU

- System
- Private peripheral bus - External
- Private peripheral bus - Internal

- Reserved
- SCS
- Reserved
- FPB
- DWT
- ITM

- External device 1.0GB

- 32MB Bit band alias
- 31MB
- 1MB Bit band region

- External RAM 1.0GB

- 32MB Bit band alias
- 31MB
- 1MB Bit band region

- Peripheral 0.5GB
- SRAM 0.5GB
- Code 0.5GB

- 0x00000000
- 0x00100000
- 0x00400000
- 0x00420000
- 0x00410000
- 0x00400000
- 0x00400000
- 0x00300000
- 0x00200000
- 0x00100000
- 0x00100000
- 0x00000000
**Instruction encoding: ADD immediate**

<table>
<thead>
<tr>
<th>Encoding T1</th>
<th>All versions of the Thumb ISA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDS &lt;Rd&gt;,&lt;Rn&gt;,#&lt;imm3&gt;</td>
<td></td>
</tr>
<tr>
<td>ADDc&lt; &lt;Rd&gt;,&lt;Rn&gt;,#&lt;imm3&gt;</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 1 1 1 0 imm3 Rn Rd</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoding T2</th>
<th>All versions of the Thumb ISA.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDS &lt;Rdn&gt;,#&lt;imm8&gt;</td>
<td></td>
</tr>
<tr>
<td>ADDc&lt; &lt;Rdn&gt;,#&lt;imm8&gt;</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>0 0 1 1 0 Rdn imm8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoding T3</th>
<th>ARMv7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD{S}&lt;,W &lt;Rd&gt;,&lt;Rn&gt;,#&lt;const&gt;</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 0 i 0 1 0 0 0 S Rn 0 imm3 Rd imm8</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Encoding T4</th>
<th>ARMv7-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDW{S} &lt;Rd&gt;,&lt;Rn&gt;,#&lt;imm12&gt;</td>
<td></td>
</tr>
<tr>
<td>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</td>
<td></td>
</tr>
<tr>
<td>1 1 1 1 0 i 1 0 0 0 0 0 Rn 0 imm3 Rd imm8</td>
<td></td>
</tr>
</tbody>
</table>
A6.7.3 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It can optionally update the condition flags based on the result.

**Encoding T1** All versions of the Thumb ISA.
ADD \(<Rd>,<Rn>,#\(<imm3>\) 
ADD\(<Cc>\ <Rd>,<Rn>,#\(<imm3>\) 

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

d = UInt(Rd); n = UInt(Rn); setflags = !InITBlock(); imm32 = ZeroExtend(imm3, 32);

**Encoding T2** All versions of the Thumb ISA.
ADD \(<Rdn>,#\(<imm8>\) 
ADD\(<Cc>\ <Rdn>,#\(<imm8>\) 

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>imm8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

d = UInt(Rdn); n = UInt(Rdn); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32);

**Encoding T3** ARMv7-M
ADD\(<S>W>\ <Rd>,<Rn>,#\(<const>\) 

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>Rn</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
</tr>
</tbody>
</table>

if Rd == '1111' & S == '1' then SEE OMN (immediate);
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd); n = UInt(Rn); setflags = (S == '1'); imm32 = ThumbExpandImm(i:imm3:imm8);
if d IN {13,15} || n == 15 then UNPREDICTABLE;

**Encoding T4** ARMv7-M
ADD\(<W>\ <Rd>,<Rn>,#\(<imm12>\) 

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Rn</td>
<td>0</td>
<td>imm3</td>
<td>Rd</td>
<td>imm8</td>
<td></td>
</tr>
</tbody>
</table>

if Rn == '1111' then SEE ADR;
if Rn == '1101' then SEE ADD (SP plus immediate);
d = UInt(Rd); n = UInt(Rn); setflags = FALSE; imm32 = ZeroExtend(i:imm3:imm8, 32);
if d IN {13,15} then UNPREDICTABLE;
<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>
# 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 Shift instructions on page A4-10 for details. Thumb permits use of a modified immediate constant or a zero-extended 16-bit immediate constant.</td>
</tr>
<tr>
<td>MVN</td>
<td>Bitwise NOT</td>
<td>Has only one operand, with the same options as the second operand in most of these instructions.</td>
</tr>
<tr>
<td>CMN</td>
<td>Bitwise OR NOT</td>
<td>-</td>
</tr>
</tbody>
</table>

Many more!
## Load/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>LDRRT</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>LDRB</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>
Addressing Modes (again)

- **Offset Addressing**
  - Offset is added or subtracted from base register
  - Result used as effective address for memory access
  - \[<Rn>, <offset>\]

- **Pre-indexed Addressing**
  - Offset is applied to base register
  - Result used as effective address for memory access
  - Result written back into base register
  - \[<Rn>, <offset>\]!

- **Post-indexed Addressing**
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
  - \[<Rn>], <offset>\]
<offset> options

• An immediate constant
  • #10

• An index register
  • <Rm>

• A shifted index register
  • <Rm>, LSL #<shift>

• Lots of weird options...
### A5.3.2 Modified immediate constants in Thumb 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 abcd efgh</td>
</tr>
<tr>
<td>0001x</td>
<td>00000000 abcd efgh 00000000 abcd efgh b</td>
</tr>
<tr>
<td>0010x</td>
<td>abcd efgh 00000000 abcd efgh 00000000 b</td>
</tr>
<tr>
<td>0011x</td>
<td>abcd efgh abcd efgh abcd efgh abcd efgh b</td>
</tr>
<tr>
<td>01000</td>
<td>1bcd efgh 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01001</td>
<td>01bcd efgh 00000000 00000000 00000000</td>
</tr>
<tr>
<td>01010</td>
<td>001bc def gh000000 00000000 00000000</td>
</tr>
<tr>
<td>01011</td>
<td>0001bc de fgh0000 00000000 00000000</td>
</tr>
</tbody>
</table>

8-bit values shifted to other positions

| 11101    | 00000000 00000000 000001bc def gh000 |
| 11110    | 00000000 00000000 000001bc cdef gh00 |
| 11111    | 00000000 00000000 00000001 bc def gh0 |

---

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

b. **UNPREDICTABLE** if abcd efgh == 00000000.
A2.3.2 The Application Program Status Register (APSР)

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

```
31 30 29 28 27 26
N  Z  C  V  Q  RESERVED
```

APSР 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 (xPSР)* 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.
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
- **ADDS**
  - $Rx = Rx + Ry$
  - APSR N, Z, C, V updated
Overflow and carry in APSR

unsigned_sum = UInt(x) + UInt(y) + UInt(carry_in);

signed_sum = SInt(x) + SInt(y) + UInt(carry_in);

result = unsigned_sum<N-1:0>; // == signed_sum<N-1:0>

carry_out = if UInt(result) == unsigned_sum then '0' else '1';

overflow = if SInt(result) == signed_sum then '0' else '1';
Conditional execution: append to many instructions for conditional execution

<table>
<thead>
<tr>
<th>cond</th>
<th>Mnemonic extension</th>
<th>Meaning (integer)</th>
<th>Meaning (floating-point) a b</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. LS (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 Assembler
(Sourcery G++ Lite 2010q1-188)
Version 2.8.5

The Free Software Foundation, Inc., inside the Free Computer Company of Australia Ltd.

Dear Elmes, Jay Franklin & friends

Richard M. Stallman and the GCC Developer Community

The GNU linker
(Sourcery G++ Lite 2010q1-188)
Version 2.8.5

Steve Chamberlain
Ina Lance Taylor

The GNU Binary Utilities
(Sourcery G++ Lite 2010q1-188)
Version 2.8.5

Roland H. Pesch
Jeffrey M. Oeler
Cygnus Support

Debugging with GDB
(Sourcery G++ Lite 2010q1-188)
Version 2.8.5

Outline

- Embedded system
- ISA
- ABI
- Build process
ABI

Need conventions for processor, compilers, linkers, and programmers to work in concert.
ABI summary

Detailed version
• Pass: r0-r3
• Return: r0 or r0-r1
• Callee saved variables: r4-r8, r11, maybe r9, r10
• Static base: r9 (might offset from this to write)
• Stack limit checking: r10 (SP >= r10)
• Veneers, scratch: r12 (lillypad)
• Stack pointer: r13
• Link register (function call return address): r14
• Program counter: r15

Simple version
• Callee preserves r4-r11 and r13
• Caller preserves r0-r3
ABI details

- A subroutine must preserve the contents of the registers r4-r8, r11, maybe r9-r10
- Arguments are passed through r0 to r3
  - If you need more, we put a pointer into memory in one of the registers.
- Return value is placed in r0 or r0-r1
- Allocate space on stack as needed. Use it as needed.
  - Reset stack pointer when done
  - Word align
Outline

- Embedded system
- ISA
- ABI
- Build process
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
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:
```

Disassembled object code

example1.out: file format elf32-littlearm

Disassembly of section .text:

00000000 <_start>:
  0: 20000800   .word 0x20000800
  4: 00000000   .word 0x00000000

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>
How does an assembly language program get turned into a executable program image?

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

as (assembler) → ld (linker) → objcopy

Memory layout → Linker script (.ld) → Disassembled code (.lst)
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?

```
arm-none-eabi-as -mcu=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
```

disassembly:
```
.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:  00000000  .word  0x00000000

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
```

example1.out: file format elf32-littlearm
Elements of assembly language program?

```assembly
.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

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 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)

Linker script (.ld)

Memory layout

Executable image file

ld (linker)

gcc (compile + link)

objc

objdump

Disassembled Code (.lst)

Binary program file (.bin)
<table>
<thead>
<tr>
<th>Register</th>
<th>Synonym</th>
<th>Special</th>
<th>Role in the procedure call standard</th>
</tr>
</thead>
<tbody>
<tr>
<td>r15</td>
<td></td>
<td>PC</td>
<td>The Program Counter.</td>
</tr>
<tr>
<td>r14</td>
<td></td>
<td>LR</td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r13</td>
<td></td>
<td>SP</td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r12</td>
<td></td>
<td>IP</td>
<td>The Intra-Procedure-call scratch register.</td>
</tr>
<tr>
<td>r11</td>
<td>v8</td>
<td></td>
<td>Variable-register 8.</td>
</tr>
<tr>
<td>r10</td>
<td>v7</td>
<td></td>
<td>Variable-register 7.</td>
</tr>
<tr>
<td>r9</td>
<td>v6</td>
<td>v6</td>
<td>Platform register.</td>
</tr>
<tr>
<td></td>
<td>SB</td>
<td>SB</td>
<td>The meaning of this register is defined by the platform standard.</td>
</tr>
<tr>
<td></td>
<td>TR</td>
<td>TR</td>
<td></td>
</tr>
<tr>
<td>r8</td>
<td>v5</td>
<td></td>
<td>Variable-register 5.</td>
</tr>
<tr>
<td>r7</td>
<td>v4</td>
<td></td>
<td>Variable register 4.</td>
</tr>
<tr>
<td>r6</td>
<td>v3</td>
<td></td>
<td>Variable register 3.</td>
</tr>
<tr>
<td>r5</td>
<td>v2</td>
<td></td>
<td>Variable register 2.</td>
</tr>
<tr>
<td>r4</td>
<td>v1</td>
<td></td>
<td>Variable register 1.</td>
</tr>
<tr>
<td>r3</td>
<td>a4</td>
<td></td>
<td>Argument / scratch register 4.</td>
</tr>
<tr>
<td>r2</td>
<td>a3</td>
<td></td>
<td>Argument / scratch register 3.</td>
</tr>
<tr>
<td>r1</td>
<td>a2</td>
<td></td>
<td>Argument / result / scratch register 2.</td>
</tr>
<tr>
<td>r0</td>
<td>a1</td>
<td></td>
<td>Argument / result / scratch register 1.</td>
</tr>
</tbody>
</table>
Done.