Design of Microprocessor-Based Systems

Prabal Dutta
University of Michigan

Modified by Jim Huang <jserv.tw@gmail.com>

Lecture: Architecture, Assembly, and ABI

Slides developed in part by
Mark Brehob
Review

• What distinguishes embedded systems?
  - Application-specific
  - Resource-constrained
  - Real-time operations
  - Physically-embodied
  - 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
Architecture
In the context of computers, what does architecture mean?
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)

32-bits

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

Endianess

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
  - 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
**ARM Cortex-M3 ISA**

### Instruction Set

ADD Rd, Rn, <op2>

### Register Set

<table>
<thead>
<tr>
<th>Register</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>R13 (SP)</td>
</tr>
<tr>
<td>R14 (LR)</td>
</tr>
<tr>
<td>R15 (PC)</td>
</tr>
<tr>
<td>xPSR</td>
</tr>
</tbody>
</table>

### Address Space

- **System**
  - Range: 0xFFFFFFFF
- **Private peripheral bus - External**
  - Range: 0xE0100000
- **Private peripheral bus - Internal**
  - Range: 0xE0040000
- **External device**
  - Range: 0xE0000000
- **External RAM**
  - Range: 0xA0000000
- **Peripheral**
  - Range: 0x60000000
- **SRAM**
  - Range: 0x40000000
- **Code**
  - Range: 0x20000000
- **0x00000000**

### Endianness

- 32-bits

---

**Endianess**
Registers

Mode dependent
Address Space

Diagram:

0xE0100000 to 0xFFFFFFFFFF
- ROM Table
- External PPB
- ETM
- TIPIU

0xE0040000 to 0xE0040000
- Reserved
- SCS
- Reserved
- FPB
- DWT
- ITM

0x44000000
- 32MB Bit band alias

0x20100000
- 1MB Bit band region

0xE0000000
- Private peripheral bus - Internal

0xE0000000
- External device 1.0GB

0xE0000000
- External RAM 1.0GB

0xE0000000
- Peripheral 0.5GB

0xE0000000
- SRAM 0.5GB

0xE0000000
- Code 0.5GB
Instruction Encoding

ADD immediate

Encoding T1: All versions of the Thumb ISA.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 0  | 1  | 1  | 1  | 0 | 0 | 1 | 0 | 1 | 1 | 0 | imm3 | Rn | Rd |

Encoding T2: All versions of the Thumb ISA.

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 0  | 0  | 1  | 1  | 0 | 0 | 1 | 1 | 0 | Rdn | imm8 |

Encoding T3: ARMv7-M

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 1  | 1  | 1  | 0  | i | 0 | 1 | 0 | 0 | 0 | 0 | S | Rn | 0 | imm3 | Rd |

Encoding T4: ARMv7-M

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|----|----|----|----|----|----|---|---|---|---|---|---|---|---|---|---|---|
| 1  | 1  | 1  | 0  | i | 1 | 0 | 0 | 0 | 0 | 0 | 0 | Rn | 0 | imm3 | Rd |

imm8 | imm8 | imm8 |
<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 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>Thumb permits use of a modified immediate constant or a zero-extended 12-bit immediate constant.</td>
</tr>
<tr>
<td>ADD</td>
<td>Add</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>ADR</td>
<td>Form PC-relative Address</td>
<td></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>CMP</td>
<td>Compare</td>
<td>Sets flags. Like ADD 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>
<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, <strong>PLDW (immediate)</strong> on page A6-176</td>
</tr>
<tr>
<td></td>
<td><strong>PLD (register)</strong> 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

• **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

| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | i | imm3 | a | b | c | d | e | f | g | h |
| \[0000x\] | 00000000 | 00000000 | 00000000 | abcdefgh |
| \[0001x\] | 00000000 | abcdefgh | 00000000 | abcdefgh |
| \[0010x\] | abcdefgh | 00000000 | abcdefgh | 00000000 |
| \[0011x\] | abcdefgh | abcdefgh | abcdefgh | abcdefgh |
| \[01000\] | 1bcdefgh | 00000000 | 00000000 | 00000000 |
| \[01001\] | 01bcdefgh | 00000000 | 00000000 | 00000000 |
| \[01010\] | 001bcdefgh | 00000000 | 00000000 | 00000000 |
| \[01011\] | 001bcdefgh | 00000000 | 00000000 | 00000000 |
| \[11101\] | 00000000 | 00000000 | 000001bc | defgh000 |
| \[11110\] | 00000000 | 00000000 | 0000001b | cdefgh000 |
| \[11111\] | 00000000 | 00000000 | 00000001 | bcdefgh0 |

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

- 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).
- UNPREDICTABLE if abcdefgh == 00000000.
Application Program Status Register (APSR)

<table>
<thead>
<tr>
<th>31</th>
<th>30</th>
<th>29</th>
<th>28</th>
<th>27</th>
<th>26</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Z</td>
<td>C</td>
<td>V</td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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
### 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) (^{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 \neq 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 \neq 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>

\(^{ab}\) For floating-point comparisons, unordered means neither equal, greater than, or less than.

\(^{c}\) Equivalent to Carry set.

\(^{d}\) Equivalent to Carry clear.

\(^{e}\) None (AL) is a special case of unconditional execution.
Exercise:
What is the value of r2 at done?

...  
start:
    movs r0, #1
    movs r1, #1
    movs r2, #1
    sub  r0, r1
    bne  done
    movs r2, #2

done:
    b    done

...
Solution:
what is the value of r2 at done?

... 

start:

movs r0, #1 // r0 ← 1, Z=0
movs r1, #1 // r1 ← 1, Z=0
movs r2, #1 // r2 ← 1, Z=0
sub r0, r1 // r0 ← r0-r1
    // but Z flag untouched
    // since sub vs subs
bne done // NE true when Z==0
    // So, take the branch
movs r2, #2 // not executed
done:
    b done // r2 is still 1
...
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
from https://github.com/embedded2013/arm-examples

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
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>
The endianess 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</th>
<th>Value</th>
<th>Offset (LSB) (MSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t a = 1;</td>
<td>0x0000 01 02 FF 00</td>
<td></td>
</tr>
<tr>
<td>uint8_t b = 2;</td>
<td>0x0004 78 56 34 12</td>
<td></td>
</tr>
<tr>
<td>uint16_t c = 255; // 0x00FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint32_t d = 0x12345678;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• **Big-Endian**
  - MSB is at lower address

<table>
<thead>
<tr>
<th>Memory</th>
<th>Value</th>
<th>Offset (LSB) (MSB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint8_t a = 1;</td>
<td>0x0000 01 02 00 FF</td>
<td></td>
</tr>
<tr>
<td>uint8_t b = 2;</td>
<td>0x0004 12 34 56 78</td>
<td></td>
</tr>
<tr>
<td>uint16_t c = 255; // 0x00FF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>uint32_t d = 0x12345678;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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>movs r1, #0</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>

```
Encoding T1
MOV5 <Rd>,<imm8>
MOVc <Rd>,<imm8>

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
0 0 1 0 0 Rd imm8

d = UInt(Rd); setflags = !InITBlock(); imm32 = ZeroExtend(imm8, 32); carry = APSR.C;
```
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) → ld (linker) → Executable image file

- as (assembler)
- Linker script (.ld)
- Memory layout
- Disassembled code (.lst)
- objdump
- objcopy
- Binary program file (.bin)
- Executable image file
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

```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 example1.bin
arm-none-eabi-objdump \
    -S example1.out > example1.lst
```
What information does the disassembled file provide?

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
```

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

equ STACK_TOP, 0x20000800
.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>
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:                   /* start label is reset handler */
  .word STACK_TOP, start
  0x20000800 /* Inserts word */

start:                   /* Inserts word (start) */
  movs r0, #10
  movs r1, #0

loop:                    /* We’ve seen the rest ... */
  adds r1, r0
  subs r0, #1
  bne  loop

deadloop:                /* Inserts word */
  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 */
    /* Specifies start is a function */

_start:  /* start label is reset handler */
    .word 0x20000800 */
    STACK_TOP, start  /* Inserts word */
    start:  /* Inserts word (start) */
        movs r0, #10
        movs r1, #0
        loop:
            adds r1, r0
            subs r0, #1
            bne loop
        deadloop:  /* We've seen the rest ... */
            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`

**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?

```bash
$ 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>NULL</td>
<td>00000000</td>
<td>0000</td>
<td>000000</td>
<td>00</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>0003</td>
<td>0001</td>
<td>00</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>0030</td>
<td>000008</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>0048</td>
<td>000000</td>
<td>00</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>0048</td>
<td>000000</td>
<td>00</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>0048</td>
<td>000021</td>
<td>00</td>
<td>WA</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>0069</td>
<td>000040</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000</td>
<td>00214</td>
<td>0000c0</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>00002c</td>
<td>00</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)
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?

```
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>NULL</td>
<td>00000000 000000 000000 00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>.text</td>
<td>PROGBITS</td>
<td>00000000 008000 000014 00</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>.ARM.attributes</td>
<td>ARM_ATTRIBUTES</td>
<td>00000000 008014 000021 00</td>
<td></td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000 008035 000031 00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000 008158 000130 10</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>.strtab</td>
<td>STRTAB</td>
<td>00000000 008288 000085 00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></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)
What are the contents of an executable’s .text segment?

- Use `readelf`’s `-x` option to hex dump a section
- 1st column shows memory address
- 2nd through 5th 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          ....
```
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

```bash
$ hexdump example1.out
0000000 457f 464c 0101 0001 0000 0000 0000 0000
0000010 0002 0028 0001 0000 0000 0000 0034 0000
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 2d33 4d 0600
0008030 070a 094d 0200 732e 6d79 6174 0062 732e
0008040 7274 6174 0062 732e 7368 7274 6174 0062
0008050 742e 7865 0074 412e 4d52 612e 7474 6972
0008060 00070 00070 00073 0000 0000 0000 0000 0000
0008070 00070 00070 00070 00070 00070 00070 00070 00070
* 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

```
$ arm-none-eabi-objcopy -O ihex example1.out "example1.hex"
$ cat example1.hex
:10000000000800200900000000A200021091801381A
:04001000FCD1FEE73A
:00000001FF
```

```
$ arm-none-eabi-objcopy -O srec example1.out "example1.srec"
$ cat example1.srec
S01000006578616D706C65312E73726563F7
S1130000000800200900000000A2000210918013816
S1070010FCD1FEE736
S9030000FC
```
Outline

• Minute quiz

• Announcements

• ARM Cortex-M3 ISA

• Assembly tool flow

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

```
C files (.c)
Object files (.o)
Assembly files (.s)
Library object files (.o)

gcc
(compile + link)

Id
(linker)

Mem
Memory layout

Linker script (.ld)

Executable image file

Binary program file (.bin)

objdump

objcopy

Disassembled Code (.lst)
```
Cheap trick: use `asm()` or `__asm()` macros to sprinkle simple assembly in standard C code!

```c
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 \[8\]

```bash
$ 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
```
Passing parameters via the stack

- **Benefits?**
- **Drawbacks?**
Passing parameters via the registers/stack

<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>PC</td>
<td></td>
<td>The Program Counter.</td>
</tr>
<tr>
<td>r14</td>
<td>LR</td>
<td></td>
<td>The Link Register.</td>
</tr>
<tr>
<td>r13</td>
<td>SP</td>
<td></td>
<td>The Stack Pointer.</td>
</tr>
<tr>
<td>r12</td>
<td>IP</td>
<td></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>SB TR</td>
<td>Platform register. The meaning of this register is defined by the platform standard.</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>
ABI Basic Rules

1. A subroutine must preserve the contents of the registers r4-r11 and SP

2. Arguments are passed though r0 to r3
   - If we need more, we put a pointer into memory in one of the registers.
     • We’ll worry about that later.

3. Return value is placed in r0
   - r0 and r1 if 64-bits.

4. Allocate space on stack as needed. Use it as needed.
   - Put it back when done...
   - Keep word aligned.
Other useful facts

- **Stack grows down.**
  - And pointed to by “SP”
- **Address we need to go back to in “LR”**

And useful things for the example

- **Assembly instructions**
  - `add` adds two values
  - `mul` multiplies two values
  - `bx` branch to register
A simple ABI routine

- \texttt{int bob(int a, int b)}
  - returns \(a^2 + b^2\)

- \textit{Instructions you might need}
  - \texttt{add} adds two values
  - \texttt{mul} multiplies two values
  - \texttt{bx} branch to register
Outline

• Minute quiz
• Announcements
• Review
• Assembly, C, and the ABI
• Memory
• Memory-mapped I/O
Memory-mapped I/O

- **The idea is really simple**
  - Instead of real memory at a given memory address, have an I/O device respond.

- **Example:**
  - Let’s say we want to have an LED turn on if we write a “1” to memory location 5.
  - Further, let’s have a button we can read (pushed or unpushed) by reading address 4.
    - If pushed, it returns a 1.
    - If not pushed, it returns a 0.
Basic example

• Discuss a basic bus protocol
  – Asynchronous (no clock)
  – Initiator and Target
  – REQ#, ACK#, Data[7:0], ADS[7:0], CMD
    • CMD=0 is read, CMD=1 is write.
    • REQ# low means initiator is requesting something.
    • ACK# low means target has done its job.
A read transaction

- **Say initiator wants to read location 0x24**
  - Initiator sets ADS=0x24, CMD=0.
  - Initiator *then* sets REQ# to low. (why do we need a delay? How much of a delay?)
  - Target sees read request.
  - Target drives data onto data bus.
  - Target *then* sets ACK# to low.
  - Initiator grabs the data from the data bus.
  - Initiator sets REQ# to high, stops driving ADS and CMD
  - Target stops driving data, sets ACK# to high terminating the transaction
Read transaction

ADS[7:0]  ??  0x24  ??

CMD

Data[7:0]  ??  0x55  ??

REQ#

ACK#
A write transaction
(write 0xF4 to location 0x31)

- Initiator sets ADS=0x31, CMD=1, Data=0xF4
- Initiator *then* sets REQ# to low.
- Target sees write request.
- Target reads data from data bus. (Just has to store in a register, need not write all the way to memory!)
- Target *then* sets ACK# to low.
- Initiator sets REQ# to high & stops driving other lines.
- Target sets ACK# to high terminating the transaction
The push-button
(if ADS=0x04 write 0 or 1 depending on button)
The push-button
(if ADS=0x04 write 0 or 1 depending on button)

Button (0 or 1)

What about CMD?