Announcements

• Saleae Logic 16
  • Sampling increases if you reduce channels
• Oscilloscope
  • Always reset to factory default when you start

• On Thursday, special session on PCB Design
  • Taught by Enoch Lee in MEB 2555
  • 12:25 - 1:45 pm
Announcements

• Books
  – The Definitive Guide to the ARM Cortex-M3
    Second Edition
  – Embedded Linux Primer
    Second Edition
  – ARM System-on-chip Architecture
    Second Edition
  – ARM Assembly Language

• Graduate Students
  – You will make presentations in second part of class
  – Your project should have a research component
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)

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

System
0xFFFFF000
0xE0100000
Private peripheral bus - External
Private peripheral bus - Internal
External device
1.0GB
0x0E040000
0x0E000000
External RAM
1.0GB
0x0A000000
0x06000000
Peripheral
0.5GB
0x40000000
0x04000000
SRAM
0.5GB
0x20000000
0x02000000
Code
0.5GB
0x00000000
0x00000000

Endianness
### 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>

LDR and LDM instructions can also cause a branch. See Load and store instructions on page A4-16 and Load/store multiple instructions on page A4-19 for details.
### Branch Encoding

#### Encoding T1

All versions of the Thumb instruction set.

**B<e> <label>**

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

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

---

#### Encoding T2

All versions of the Thumb instruction set.

**B<e> <label>**

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>imm11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 0 0</td>
<td></td>
</tr>
</tbody>
</table>

imm32 = SignExtend(imm11:'0', 32);
if InITBlock() && !LastInITBlock() then UNPREDICTABLE;

---

#### Encoding T3

ARMv7-M

**B<e>.W <label>**

Not allowed in IT block.

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>cond</th>
<th>imm6</th>
<th>1 0 1 J1 0 J2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0</td>
<td>S</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

#### Encoding T4

ARMv7-M

**B<e>.W <label>**

Outside or last in IT block

<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
<th>imm10</th>
<th>1 0 1 J1 1 J2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 0</td>
<td>S</td>
<td></td>
</tr>
</tbody>
</table>

I1 = NOT(J1 EOR S);  I2 = NOT(J2 EOR S);  imm32 = SignExtend(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 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 ITE Instruction

• Useful for small If-Then-Else constructs

• IT<x><y><z> <cond>

– <x>, <y>, <z> can be T or E which refers to the base condition <cond>

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

CMP R0, R1 ; Compare R0 and R1
ITTTEE 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:

CBZ <Rn>, <label>
or:
CBNZ <Rn>, <label>
where label is a forward branch address. For example:

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

This can be written as:

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

Flags are not affected by this instruction.
Instruction classes

- Branching
- Data processing
- Load/store
- Exceptions
- Miscellaneous
Addressing Modes

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

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

• Post-indexed Addressing: [<Rn>], <offset>
  - The address from the base register is used as the EA
  - The offset is applied to the base and then written back
• An immediate constant
  - #10

• An index register
  - <Rm>

• A shifted index register
  - <Rm>, LSL #<shift>
Updating the Application Program Status Register (aka condition codes or APRS)

• sub r0, r1
  - r0 ← r0 – r1
  - APSR remain unchanged

• subs r0, r1
  - r0 ← r0 – r1
  - APSR N or Z bits could change

• add r0, r1
  - r0 ← r0 + r1
  - APSR remain unchanged

• adds r0, r1
  - r0 ← r0 + r1
  - APSR C or V bits could change
```assembly
; 0000017c <main>:
17c:   b580   push  {r7, lr}
17e:   b084   sub   sp, #16
180:   af00   add   r7, sp, #0
182:   f04f 0328  mov.w r3, #40 ; 0x28
186:   60bb   str   r3, [r7, #8]
188:   f04f 0300  mov.w r3, #0
18c:   60fb   str   r3, [r7, #12]
18e:   f04f 0300  mov.w r3, #0
192:   603b   str   r3, [r7, #0]
194:   e010   b.n   1b8 <main+0x3c>
196:   6838   ldr   r0, [r7, #0]
198:   f7ff ffb8  bl   10c <factorial>
19c:   4603   mov   r3, r0
19e:   607b   str   r3, [r7, #4]
1a0:   f646 5010  movw  r0, #27920 ; 0x6d10
1a4:   f2c0 0000  movt  r0, #0
1a8:   6839   ldr   r1, [r7, #0]
1aa:   687a   ldr   r2, [r7, #4]
1ac:   f000 f840  bl   230 <printf>
1b0:   683b   ldr   r3, [r7, #0]
...```
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

- **Big-Endian**
  - MSB is at lower address
Instruction encoding

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

How?

Instructions

| movs r0, #10 | movs r1, #0 |

Register Value

<table>
<thead>
<tr>
<th>001</th>
<th>00</th>
<th>000</th>
<th>00001010</th>
</tr>
</thead>
<tbody>
<tr>
<td>(msb)</td>
<td>(lsb)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Memory Value

<table>
<thead>
<tr>
<th>0a 20 00 21</th>
</tr>
</thead>
</table>

Encoding T1

All versions of the Thumb instruction set.

ARMv7 ARM

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

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

| 0 0 1 0 0 | Rd |

| 0 0 1 0 0 | imm8 |
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

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

_start:

.start:

.word                  STACK_TOP, start

.start:                 movs r0, #10

...                  
```
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 an executable program image?

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

Memory layout
Linker script (.ld)

Binary program file (.bin)
Disassembled code (.lst)

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?

```assembly
.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>
```
What are the elements of a **real** assembly 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 */
                           /* 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
```
Exercise: How often do we execute the loop?

```
.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 label is reset handler */

.type start, %function /* We’ve seen the rest ... */

_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 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?

```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>000000</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>000034</td>
<td>000014</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>000300</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>000048</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>000048</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>000048</td>
<td>000021</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>000069</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>000214</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>0000d4</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:

<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>000000</td>
<td>000000</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>008000</td>
<td>000014</td>
<td>0</td>
<td>AX</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>.ARM.attributes</td>
<td>ARM_ATTRIBUTES</td>
<td>00000000</td>
<td>008014</td>
<td>000021</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>.shstrtab</td>
<td>STRTAB</td>
<td>00000000</td>
<td>008035</td>
<td>000031</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>.symtab</td>
<td>SYMTAB</td>
<td>00000000</td>
<td>008158</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>008288</td>
<td>000085</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</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

$\texttt{readelf -x .text example1.out}$

\textbf{Hex dump of section ‘.text’}:
\begin{verbatim}
  0x00000000 | 00080020 09000000 0a200021 09180138 ... ... .!...8
  0x00000010 | fcd1fee7
\end{verbatim}
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 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 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
:1000000000000000800200900000000A200021091801381A
:04001000FCD1FEE73A
:00000001FF
```

```bash
$ 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 an assembly language program get turned into an executable program image?

- **Assembly files (.s)**
  - **Object files (.o)** (assembler)
  - **Executable image file** (linker)
  - **Memory layout**
  - **Linker script (.ld)**
  - **Disassembled code (.lst)**
  - **Binary program file (.bin)**

The diagram shows the process of transforming an assembly language program into an executable program image using the following tools:

- **objcopy**
- **objdump**
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)**

Steps:
1. **as** (assembler) processes Assembly files (.s) to create Object files (.o).
2. **gcc** (compile + link) processes C files (.c) and Object files (.o) to create an executable image file.
3. **ld** (linker) processes the executable image file, along with Memory layout and Linker script (.ld), to create the final executable program image.
4. **objcopy** creates a binary program file (.bin).
5. **objdump** disassembles the code to create a Disassembled code (.lst).
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!
Questions?

Comments?

Discussion?
## 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!**
## Miscellaneous instructions

Table A4-12 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>
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.