Announcements

• Homework 1 due tonight!
• Homework 2 next week
  • ARM Assembly
  • ABI
• Use the Etherpad
  • password: ece5780pw

Inline Assembly
• See http://www.ethernut.de/en/documents/arm-inline-asm.html
• \texttt{asm(code : output list : input list : clobber list);}  
• \texttt{asm("mov %[result], %[value], ror #1" : [result] ":=r" (y) : [value] ":r" (x));}
• Pre 3.1: \texttt{asm("mov %0, %1, ror #1" : ":=r" (result) : ":r" (value));}
• `asm volatile(`
  "ands    r3, %1, #3\n\t`
  "eor     %0, %0, r3\n\t`
  "addne   %0, #4"
  : "=r" (len)
  : "0" (len)
  : "cc", "r3"
`);

• Constraints;
  – r: General Register
  – 0: Use the same input register as for first output operand

• Modifier:
  – =: write-only
  – +: read-write
  – &: register should be used for output only
The Dreaded Minute Quiz
Outline

• Minute quiz

• Announcements

• Review

• Assembly, C, and the ABI

• Memory

• Memory-mapped I/O
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**

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

_start:

.start:
.word STACK_TOP, start

movs r0, #10
...
```

![Diagram of memory map with SRAM and Code segments](image)
Major elements of an Instruction Set Architecture
(registers, memory, word size, endianess, conditions, instructions, addressing modes)

32-bits

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

xPSR

Endianness

32-bits

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

Endianness

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

- Instructions are encoded in machine language opcodes
- Sometimes
  - Distinguish opcodes from each other
  - Necessary to decode opcodes and itemize arch state impacts

How?

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

Encoding T1

All versions of the Thumb instruction set.

MOVS <Rd>,#<imm8>
MOV<c> <Rd>,#<imm8>
Thumb instructions are a sequence of half-word-aligned half-words

Each Thumb instruction is either
- a 16-bit half-word in that stream
- A 32-bit instruction consisting of two half-words in that stream

If bits [15:11] of the half-word being decoded take on any of the following values
- 0b11101
- 0b11110
- 0b11111
- then half-word is the first half-word of a 32-bit instruction
- otherwise the half-word is a 16-bit instruction

See ARM ARM A5.1, A5.5, A5-13
16-bit Thumb instruction encoding

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction or instruction class</th>
</tr>
</thead>
<tbody>
<tr>
<td>00xxxx</td>
<td>Shift (immediate), add, subtract, move, and compare on page A5-6</td>
</tr>
<tr>
<td>010000</td>
<td>Data processing on page A5-7</td>
</tr>
<tr>
<td>010001</td>
<td>Special data instructions and branch and exchange on page A5-8</td>
</tr>
<tr>
<td>0101xx</td>
<td>Load from Literal Pool, see LDR (literal) on page A6-90</td>
</tr>
<tr>
<td>0101xx</td>
<td>Load/store single data item on page A5-9</td>
</tr>
<tr>
<td>011xxx</td>
<td></td>
</tr>
<tr>
<td>100xxx</td>
<td></td>
</tr>
<tr>
<td>10100x</td>
<td>Generate PC-relative address, see ADR on page A6-30</td>
</tr>
<tr>
<td>10101x</td>
<td>Generate SP-relative address, see ADD (SP plus immediate) on page A6-26</td>
</tr>
<tr>
<td>1011xx</td>
<td>Miscellaneous 16-bit instructions on page A5-10</td>
</tr>
<tr>
<td>11000x</td>
<td>Store multiple registers, see STM / STMIA / STMEA on page A6-218</td>
</tr>
<tr>
<td>11001x</td>
<td>Load multiple registers, see LDM / LDMIA / LDMFD on page A6-84</td>
</tr>
<tr>
<td>1101xx</td>
<td>Conditional branch, and supervisor call on page A5-12</td>
</tr>
<tr>
<td>11100x</td>
<td>Unconditional Branch, see B on page A6-40</td>
</tr>
</tbody>
</table>
Shift (immediate), add, subtract, move, and compare

<table>
<thead>
<tr>
<th>opcode</th>
<th>Instruction</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>000xx</td>
<td>Logical Shift Left</td>
<td><em>LSL (immediate)</em> on page A6-134</td>
</tr>
<tr>
<td>001xx</td>
<td>Logical Shift Right</td>
<td><em>LSR (immediate)</em> on page A6-138</td>
</tr>
<tr>
<td>010xx</td>
<td>Arithmetic Shift Right</td>
<td><em>ASR (immediate)</em> on page A6-36</td>
</tr>
<tr>
<td>01100</td>
<td>Add register</td>
<td><em>ADD (register)</em> on page A6-24</td>
</tr>
<tr>
<td>01101</td>
<td>Subtract register</td>
<td><em>SUB (register)</em> on page A6-246</td>
</tr>
<tr>
<td>01110</td>
<td>Add 3-bit immediate</td>
<td><em>ADD (immediate)</em> on page A6-22</td>
</tr>
<tr>
<td>01111</td>
<td>Subtract 3-bit immediate</td>
<td><em>SUB (immediate)</em> on page A6-244</td>
</tr>
<tr>
<td>100xx</td>
<td>Move</td>
<td><em>MOV (immediate)</em> on page A6-148</td>
</tr>
<tr>
<td>101xx</td>
<td>Compare</td>
<td><em>CMP (immediate)</em> on page A6-62</td>
</tr>
<tr>
<td>110xx</td>
<td>Add 8-bit immediate</td>
<td><em>ADD (immediate)</em> on page A6-22</td>
</tr>
<tr>
<td>111xx</td>
<td>Subtract 8-bit immediate</td>
<td><em>SUB (immediate)</em> on page A6-244</td>
</tr>
</tbody>
</table>

Table A5-2 shows the allocation of encodings in this space.

*Table A5-2 16-bit shift(immediate), add, subtract, move and compare encoding*
OUTPUT_FORMAT("elf32-littlearm")
OUTPUT_ARCH(arm)
ENTRY(main)

MEMORY
{
/* SmartFusion internal eSRAM */
  ram (rwx) : ORIGIN = 0x20000000, LENGTH = 64k
}

SECTIONS
{
.text :
{
  . = ALIGN(4);
  *(.text*)
  . = ALIGN(4);
  _etext = .;
} >ram
}
end = .;

- Specifies little-endian arm in ELF format.
- Specifies ARM CPU
- Should start executing at label named “main”
- We have 64k of memory starting at 0x20000000. You can read (r), write (w) and execute (x) out of it. We’ve named it “ram”

- “.” is a reference to the current memory location
- First align to a word (4 byte) boundary
- Place all sections that include .text at the start (* here is a wildcard)
- Define a label named _etext to be the current address.
- Put it all in the memory location defined by the ram memory location.
Some things to think about (TTTA)

- What instruction set? Thumb!
- What is conditional execution (ARM ARM, A4.1.2)?
- What are the side effects of instruction execution?
How does an assembly language program get turned into a executable program image?

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

- Assembly files (.s) → Object files (.o) with `as` (assembler)
- Object files (.o) → Executable image file with `ld` (linker)

Additional tools and files:
- `objcopy` to convert binary program file (.bin) to executable image file
- `objdump` to disassemble code (.lst)
- Memory layout
- Linker script (.ld)
Outline

• Minute quiz

• Announcements

• Review

• Assembly, C, and the ABI

• Memory

• Memory-mapped I/O
How does a mixed C/Assembly program get turned into a executable program image?

1. C files (.c)
2. Assembly files (.s)
3. Object files (.o)
4. Library object files (.o)
5. Memory layout
6. Linker script (.ld)
7. Executable image file
8. ld (linker)
9. gcc (compile + link)
10. Binary program file (.bin)
11. objcopy
12. objdump
13. Disassembled Code (.lst)
• 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></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 SB TR</td>
<td></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>
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
  - mla multiply and accumulate
  - bx branch to register
A simple ABI routine

- int bob(int a, int b)
  - returns $a^2 + b^2$

- Instructions you might need
  - add  adds two values
  - mul  multiplies two values
  - mla  multiply and accumulate
  - bx   branch to register
• int bob(int a, int b)
  - returns $a^2 + b^2$
Some disassembly

- 0x20000490 <bob>: push {r7}
- 0x20000492 <bob+2>: sub sp, #20
- 0x20000494 <bob+4>: add r7, sp, #0
- 0x20000496 <bob+6>: str r0, [r7, #4]
- 0x20000498 <bob+8>: str r1, [r7, #0]
- x=a*a;
- 0x2000049a <bob+10>: ldr r3, [r7, #4]
- 0x2000049c <bob+12>: ldr r2, [r7, #4]
- 0x2000049e <bob+14>: mul.w r3, r2, r3
- 0x200004a2 <bob+18>: str r3, [r7, #8]
- y=b*b;
- 0x200004a4 <bob+20>: ldr r3, [r7, #0]
- 0x200004a6 <bob+22>: ldr r2, [r7, #0]
- 0x200004a8 <bob+24>: mul.w r3, r2, r3
- 0x200004ac <bob+28>: str r3, [r7, #12]
- x=x+y;
- 0x200004ae <bob+30>: ldr r2, [r7, #8]
- 0x200004b0 <bob+32>: ldr r3, [r7, #12]
- 0x200004b2 <bob+34>: add r3, r2
- 0x200004b4 <bob+36>: str r3, [r7, #8]
- return(x);
- 0x200004b6 <bob+38>: ldr r3, [r7, #8]
- }
- 0x200004b8 <bob+40>: mov r0, r3
- 0x200004ba <bob+42>: add.w r7, r7, #20
- 0x200004be <bob+46>: mov sp, r7
- 0x200004c0 <bob+48>: pop {r7}
- 0x200004c2 <bob+50>: bx lr

```c
int bob(int a, int b)
{
    int x, y;
    x=a*a;
    y=b*b;
    x=x+y;
    return(x);
}
```
Outline

- Minute quiz
- Announcements
- Review
- Assembly, C, and the ABI
- Memory
- Memory-mapped I/O
<table>
<thead>
<tr>
<th>Processor Region</th>
<th>Memory Map of Cortex-M3</th>
<th>Memory Map of FPGA Fabric Master, Ethernet MAC, Peripheral DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Registers</td>
<td></td>
<td>0xE0043000 – 0xFFFFFFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xE0042000 – 0xE0042FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x78000000 – 0x78001FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x74000000 – 0x74001FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x70000000 – 0x70001FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x601D0000 – 0x601D1FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x60180000 – 0x60181FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x60100000 – 0x60101FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x60088200 – 0x60089FFF</td>
</tr>
<tr>
<td>External Memory Type 1</td>
<td></td>
<td>0x60088800 – 0x600891FF</td>
</tr>
<tr>
<td>External Memory Type 0</td>
<td></td>
<td>0x6008A000 – 0x6008A7FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x6008B000 – 0x6008B7FF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x6008D000 – 0x6008D7FF</td>
</tr>
<tr>
<td>eNVM Controller</td>
<td></td>
<td>0x60088000 – 0x600881FF</td>
</tr>
<tr>
<td>eNVM Controller</td>
<td></td>
<td>0x60088800 – 0x600881FF</td>
</tr>
<tr>
<td>eNVM Aux Block (spare pages)</td>
<td></td>
<td>0x6008A000 – 0x6008A7FF</td>
</tr>
<tr>
<td>eNVM Aux Block (array)</td>
<td></td>
<td>0x6008D000 – 0x6008D7FF</td>
</tr>
<tr>
<td>eNVM Spare Pages</td>
<td>eNVM Spare Pages</td>
<td>0x6008D000 – 0x6008D7FF</td>
</tr>
<tr>
<td>eNVM Spare Pages</td>
<td>eNVM Spare Pages</td>
<td>0x6008D000 – 0x6008D7FF</td>
</tr>
<tr>
<td>eNVM Array</td>
<td>eNVM Array</td>
<td>0x6008D000 – 0x6008D7FF</td>
</tr>
<tr>
<td>Peripherals (BB view)</td>
<td></td>
<td>0x40000000 – 0x40000FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x40017000 – 0x40017FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x40030004 – 0x4003000F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x400017000 – 0x400017FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x400015000 – 0x400015FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x40014000 – 0x40014FFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x40013000 – 0x40013FFF</td>
</tr>
<tr>
<td>FPGA Fabric</td>
<td>FPGA Fabric</td>
<td>0x40004000 – 0x40004FFF</td>
</tr>
<tr>
<td>Analog Compute Engine</td>
<td>Analog Compute Engine</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>IAP Controller</td>
<td>IAP Controller</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>eFROM</td>
<td>eFROM</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>RTC</td>
<td>RTC</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>MSS GPIO</td>
<td>MSS GPIO</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>I2C_1</td>
<td>I2C_1</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>SPI_1</td>
<td>SPI_1</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>UART_1</td>
<td>UART_1</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>eSRAM_0 / eSRAM_1 (BB view)</td>
<td>eSRAM_0 / eSRAM_1 (BB view)</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>eSRAM_1</td>
<td>eSRAM_1</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>eSRAM_0</td>
<td>eSRAM_0</td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>eNVM (Cortex-M3) Virtual View</td>
<td></td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
<tr>
<td>eNVM (fabric) Virtual View</td>
<td></td>
<td>0xE0043000 – 0xFFFF2FFF</td>
</tr>
</tbody>
</table>

Visible only to FPGA Fabric Master: 0xE0043000 – 0xFFFF2FFF

Figure 2-4 • System Memory Map with 64 Kbytes of SRAM
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.
Now...

• How do you get that to happen?
  - We could just say “magic” but that’s not very helpful.
  - Let’s start by detailing a simple bus and hooking hardware up to it.

• We’ll work on a real bus next time!
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
ADS[7:0] 0x24
CMD
Data[7:0] 0x55
REQ#
ACK#

A B C D E F G HI
- 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)

Button (0 or 1)
The push-button
(if ADS=0x04 write 0 or 1 depending on button)

| ADS[7] | - |
| ADS[6] | - |
| ADS[5] | - |
| ADS[4] | - |
| ADS[3] | - |
| ADS[2] | - |
| ADS[1] | - |
| ADS[0] | - |
| REQ#   | - |

ACK#
The LED
(1 bit reg written by LSB of address 0x05)

Flip-flop which controls LED
Questions?

Comments?

Discussion?