CS/ECE 5780/6780
Embedded Systems Design

Lecture 4: Review, Simulation, ABI, and Memory-Mapped I/O

Thomas Schmid
thomas.schmid@utah.edu

Adapted from Prabal Dutta (prabal@umich.edu)
Announcements

• Homework 1.2 due tonight!
  • Once corrected, submit your files to Circuit Graphics by Friday!

• Homework 2 Preview
  • ARM Assembly
  • ABI

• 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));}

• Energy Micro: Intro to C, Part 2:
  • http://blog.energymicro.com/2013/01/22/719/
• \texttt{asm volatile}(
  "\texttt{ands \quad r3, \%1, \#3}\n\texttt{\textbackslash n\textbackslash t}\"
  "\texttt{eor \quad \%0, \%0, r3}\n\texttt{\textbackslash n\textbackslash t}\"
  "\texttt{addne \quad \%0, \#4}\"
  : "\texttt{=}r" (len)
  : "\texttt{0}" (len)
  : "\texttt{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
• push {reg_list}: reg_list is stored on the stack in
  – Numerical order
  – With lowest numbered register at lowest memory address
• pop {reg_list}: reg_list is stored on the stack in
  – Numerical order
  – With lowest numbered register at lowest memory address
  – If PC is in reg_list, causes branch to addr popped off stack

• Examples
  – push {r0, r4-r7}
  – push {r2, lr}
  – pop {r0, r1}

• ARM stack is full-descending so a push causes the SP to hold a lower address.

if ConditionPassed() then
  EncodingSpecificOperations();
  address = SP - 4*BitCount(registers);
  for i = 0 to 14
    if registers<i> == ‘1’ then
      MemA[address,4] = R[i];
      address = address + 4;
  SP = SP - 4*BitCount(registers);
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:
.word STACK_TOP, start

start:
movs r0, #10
...```

![Graph showing memory layout with SRAM and Code sections]
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>movs r1, #0</td>
<td>001</td>
<td>00</td>
</tr>
</tbody>
</table>

Encoding T1

All versions of the Thumb instruction set.

ARMv7 ARM

<table>
<thead>
<tr>
<th>rd IMM8</th>
<th>value</th>
<th>format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 1 0 0</td>
<td>Rd</td>
<td>imm8</td>
</tr>
</tbody>
</table>
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

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

<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>01001x</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>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>00</td>
<td>0</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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**

<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>$LSL$ (immediate) on page A6-134</td>
</tr>
<tr>
<td>001xx</td>
<td>Logical Shift Right</td>
<td>$LSR$ (immediate) on page A6-138</td>
</tr>
<tr>
<td>010xx</td>
<td>Arithmetic Shift Right</td>
<td>$ASR$ (immediate) on page A6-36</td>
</tr>
<tr>
<td>01100</td>
<td>Add register</td>
<td>$ADD$ (register) on page A6-24</td>
</tr>
<tr>
<td>01101</td>
<td>Subtract register</td>
<td>$SUB$ (register) on page A6-246</td>
</tr>
<tr>
<td>01110</td>
<td>Add 3-bit immediate</td>
<td>$ADD$ (immediate) on page A6-22</td>
</tr>
<tr>
<td>01111</td>
<td>Subtract 3-bit immediate</td>
<td>$SUB$ (immediate) on page A6-244</td>
</tr>
<tr>
<td>100xx</td>
<td>Move</td>
<td>$MOV$ (immediate) on page A6-148</td>
</tr>
<tr>
<td>101xx</td>
<td>Compare</td>
<td>$CMP$ (immediate) on page A6-62</td>
</tr>
<tr>
<td>110xx</td>
<td>Add 8-bit immediate</td>
<td>$ADD$ (immediate) on page A6-22</td>
</tr>
<tr>
<td>111xx</td>
<td>Subtract 8-bit immediate</td>
<td>$SUB$ (immediate) on page A6-244</td>
</tr>
</tbody>
</table>
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) boundry
• 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 an executable program image?

Assembly files (.s) → Object files (.o) → (assembler) → Memory layout → Linker script (.ld) → ld (linker) → Executable image file → objcopy → Binary program file (.bin) → objdump → Disassembled code (.lst)
How does a mixed C/Assembly program get turned into a executable program image?

- Assembly files (.s)
- C files (.c)
- Library object files (.o)
- Object files (.o)
- Linker script (.ld)
- Memory layout

**Compilers:**
- GCC (compile + link)
- AS (assembler)

**Tools:**
- LD (linker)
- OBJCOPY
- OBJDUMP
- Binary program file (.bin)

**Disassembled code (.lst)**
Outline

• Minute quiz

• Announcements

• Review

• Assembly, C, and the ABI

• Memory

• Memory-mapped I/O
What is a function in C?
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></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>v9</td>
<td></td>
<td>v6 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>

Table 2: Core registers of M6800.
1. A subroutine must preserve the contents of the registers r4-r11 and SP

2. Arguments are passed through 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 $a^2 + b^2$
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>System Memory Map</th>
<th>Memory Map of FPGA Fabric Master, Ethernet MAC, Peripheral DMA</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Registers</td>
<td>0xFE043000 – 0xFFFFFFFF</td>
</tr>
<tr>
<td>External Memory Type 1</td>
<td>0xE042000 – 0xE042FFFF</td>
</tr>
<tr>
<td>External Memory Type 0</td>
<td>0x78000000 – 0xE041FFFF</td>
</tr>
<tr>
<td>eNVM Controller</td>
<td>0x74000000 – 0x77FFFFFF</td>
</tr>
<tr>
<td>eNVM Aux Block (spare pages)</td>
<td>0x70000000 – 0x73FFFFFF</td>
</tr>
<tr>
<td>eNVM Aux Block (array)</td>
<td>0x601D0000 – 0x601FFFFF</td>
</tr>
<tr>
<td>eNVM Spare Pages</td>
<td>0x60180000 – 0x601CFFFF</td>
</tr>
<tr>
<td>eNVM Array</td>
<td>0x60100100 – 0x6017FFFF</td>
</tr>
<tr>
<td>Peripherals (BB view)</td>
<td>0x60100000 – 0x601000FF</td>
</tr>
<tr>
<td>FPGA Fabric</td>
<td>0x60088200 – 0x600881FF</td>
</tr>
<tr>
<td>FPGA Fabric eSRAM Backdoor</td>
<td>0x60088000 – 0x6008881FF</td>
</tr>
<tr>
<td>APB Extension Register</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>eSRAM_0 / eSRAM_1 (BB view)</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Analog Compute Engine</td>
<td>0x60080000 – 0x600881FF</td>
</tr>
<tr>
<td>IAP Controller</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>eFROM</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>RTC</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>MSS GPIO</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>I2C_1</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>SPI_1</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>UART_1</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Fabric Interface Interrupt Controller</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Watchdog</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Timer</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Peripheral DMA</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Ethernet MAC</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>I2C_0</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>SPI_0</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>UART_0</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>eSRAM_0</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>eSRAM_1</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>eNVM (Cortex-M3) Virtual View</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>eNVM (fabric) Virtual View</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
<tr>
<td>Visible only to FPGA Fabric Master</td>
<td>0x60080000 – 0x6008081FF</td>
</tr>
</tbody>
</table>

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
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)

ADS[7]
ADS[6]
ADS[5]
ADS[4]
ADS[3]
ADS[2]
ADS[1]
ADS[0]

REQ#

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

<table>
<thead>
<tr>
<th>ADS[7]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADS[6]</td>
<td></td>
</tr>
<tr>
<td>ADS[5]</td>
<td></td>
</tr>
<tr>
<td>ADS[4]</td>
<td></td>
</tr>
<tr>
<td>ADS[3]</td>
<td></td>
</tr>
<tr>
<td>ADS[2]</td>
<td></td>
</tr>
<tr>
<td>ADS[1]</td>
<td></td>
</tr>
<tr>
<td>ADS[0]</td>
<td></td>
</tr>
<tr>
<td>REQ#</td>
<td></td>
</tr>
</tbody>
</table>
The LED
(1 bit reg written by LSB of address 0x05)
Questions?

Comments?

Discussion?