Lecture Topics

- Today: Assembly Language Basics
  (H&H 6.1-6.2)
- Next: continued

Announcements

- Self-study Module #6
- Project #5 (due no later than 6/6)
Project #5

- Major themes:
  - programming in C (using "gcc")
  - integer bitwise operations
  - floating point arithmetic
- Library module (set of functions)
- Driver module (test bed)

How to work with same 32 bits as both a floating point value and an integer value?
- union

How to unpack bit fields?
- shifting and masking (with AND)

How to pack bit fields back together?
- shifting and masking (with OR)
Project #5 (union)

union sp_object
{
    float frep;
    unsigned int irep;
};

union sp_object num;
num.frep = 24.5;

printf( "Real: %f Hex: %08x\n", num.frep, num.irep );

Real: 24.500000 Hex: 41c40000

Project #5 (unpacking)

unsigned int value = 0x6789ABCD;
unsigned int field1;
unsigned int field2;

field1 = (value & 0x00FF0000) >> 16;
field2 = (value & 0x0000FF00) >> 8;

printf( "value: %08x\n", value ); value: 6789abcd
printf( "field1: %08x\n", field1 ); field1: 00000089
printf( "field2: %08x\n", field2 ); field2: 000000ab
Project #5 (packing)

```c
unsigned int value1 = 0x1234;
unsigned int value2 = 0xABCD;
unsigned int result;

result = (value1 << 16) | value2;

printf( "value1:  %08x\n", value1 );  \textbf{value1:  00001234}
printf( "value2:  %08x\n", value2 );  \textbf{value2:  0000abcd}
printf( "result:  %08x\n", result );  \textbf{result:  1234abcd}
```

Review: The System Bus Model

Communication among components is handled by a shared pathway called the system bus, which is made up of the data bus, the address bus, and the control bus. There is also a power bus, and some architectures may also have a separate I/O bus.
ARM Model

I/O module: handles all input and output (human oriented and secondary storage)

Memory module: $2^{32}$ bytes of RAM

CPU module:
- Control Unit
- Integer Unit
- Floating Point Unit
- Coprocessor (optional)

ARM Microprocessor

RAM: holds program -- machine instructions and data
Control Unit: manages all other units
Integer Unit: circuits for integer operations
Floating Point Unit: circuits for floating point operations
Coprocessor: specialized circuits (optional)
ARM Control Unit

Manages the fetch-execute cycle, controls the other modules.

Control Unit Registers (32 bits):

  PC – address of current instruction
  IR – bit pattern of current instruction

ARM Fetch-Execute Cycle

Fetch Phase:
  \[ \text{RAM}[PC] \rightarrow IR \]

Execute Phase:
  decode IR
  take appropriate action
  update PC (usually \( PC + 4 \rightarrow PC \))
ARM Integer Unit

Tightly coupled with Control Unit.

Handles all integer calculations.

Integer Unit Registers (32 bits):

General purpose registers (r0 – r15)
CPSR (current program status register):
  includes NZCV status bits

Contents of the general-purpose registers (r0-r15), as well as CPSR at some instant:

<table>
<thead>
<tr>
<th>IU REGISTERS</th>
<th>cpsr: 40000010</th>
<th>icc: 4 (Z--)</th>
</tr>
</thead>
<tbody>
<tr>
<td>r00: 00000000</td>
<td>r04: 000219d4</td>
<td>r08: 0000000f</td>
</tr>
<tr>
<td>r01: ffffffff</td>
<td>r05: 000219e8</td>
<td>r09: 00000017</td>
</tr>
<tr>
<td>r02: 00000000</td>
<td>r06: 00000000</td>
<td>r10: 00000012</td>
</tr>
<tr>
<td>r03: 00000000</td>
<td>r07: 00000000</td>
<td>r11: 0000001d</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r12: 000219d4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r13: 000219d8</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r14: 000219dc</td>
</tr>
<tr>
<td></td>
<td></td>
<td>r15: 000103e8</td>
</tr>
</tbody>
</table>
Translation and Execution

Let's look at a very simple C program and discuss the steps required to translate the program from C into ARM machine language, then load and execute that machine language program.
The text file “simple.c” resides on disk. Run “gcc” to translate and link that program, creating the executable file “a.out” (which also resides on disk):

```
<2 lemon: > gcc -Wall simple.c
```

Load “a.out” to execute the program: have the operating system copy the contents of “a.out” into RAM, set the program counter to the first machine language instruction in the program, then let the fetch-execute cycle take over:

```
<3 lemon: > a.out
```
The machine language instructions from “a.out” after the file is loaded into memory.

Addresses are 32 bits:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001040c</td>
<td>e1a00003</td>
</tr>
</tbody>
</table>

Machine language instructions are 32 bits:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000103e8</td>
<td>e52db004</td>
</tr>
<tr>
<td>000103ec</td>
<td>e28db000</td>
</tr>
<tr>
<td>000103f0</td>
<td>e59f3024</td>
</tr>
<tr>
<td>000103f4</td>
<td>e5932000</td>
</tr>
<tr>
<td>000103f8</td>
<td>e59f3020</td>
</tr>
<tr>
<td>000103fc</td>
<td>e5933000</td>
</tr>
<tr>
<td>00010400</td>
<td>e083003</td>
</tr>
<tr>
<td>00010404</td>
<td>e59f2018</td>
</tr>
<tr>
<td>00010408</td>
<td>e5823000</td>
</tr>
<tr>
<td>0001040c</td>
<td>e1a00003</td>
</tr>
<tr>
<td>00010410</td>
<td>e24bd000</td>
</tr>
<tr>
<td>00010414</td>
<td>e49db004</td>
</tr>
<tr>
<td>00010418</td>
<td>e12fff1e</td>
</tr>
<tr>
<td>0001041c</td>
<td>000205c0</td>
</tr>
<tr>
<td>00010420</td>
<td>000205c4</td>
</tr>
<tr>
<td>00010424</td>
<td>000205c8</td>
</tr>
</tbody>
</table>

The variables (and initial values) from “a.out” after the file is loaded into memory.

Addresses are 32 bits:

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000205c0</td>
<td>0000000c</td>
</tr>
<tr>
<td>000205c4</td>
<td>0000001b</td>
</tr>
<tr>
<td>000205c8</td>
<td>ffffffff</td>
</tr>
</tbody>
</table>
Outline for function "main":

Get address of A
Load value of A
Get address of B
Load value of B
Add value of A, value of B
Get address of C
Store result at C

Two segments:

000103e8: e52db004
000103ec: e28db000
000103f0: e59f3024
000103f4: e5932000
000103f8: e59f3020
000103fc: e5933000
00010400: e0823003
00010404: e59f2018
00010408: e5823000
0001040c: e1a00003
00010410: e24bd000
00010414: e49db004
00010418: e12fff1e
0001041c: 000205c0
00010420: 000205c4
00010424: 000205c8

000205c0: 0000000c
000205c4: 0000001b
000205c8: ffffffff

Two segments:

0001xxxx: instructions
0002xxxx: data
<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>000103e8</td>
<td>e52db004 push {fp}</td>
</tr>
<tr>
<td>000103ec</td>
<td>e28db000 add fp, sp, #0</td>
</tr>
<tr>
<td>000103f0</td>
<td>e59f3024 ldr r3, [pc, #36]</td>
</tr>
<tr>
<td>000103f4</td>
<td>e5932000 ldr r2, [r3]</td>
</tr>
<tr>
<td>000103f8</td>
<td>e59f3020 ldr r3, [pc, #32]</td>
</tr>
<tr>
<td>000103fc</td>
<td>e5933000 ldr r3, [r3]</td>
</tr>
<tr>
<td>00010400</td>
<td>e0823003 add r3, r2, r3</td>
</tr>
<tr>
<td>00010404</td>
<td>e59f2018 ldr r2, [pc, #24]</td>
</tr>
<tr>
<td>00010408</td>
<td>e5823000 str r3, [r2]</td>
</tr>
<tr>
<td>0001040c</td>
<td>e1a00003 mov r0, r3</td>
</tr>
<tr>
<td>00010410</td>
<td>e24bd000 sub sp, fp, #0</td>
</tr>
<tr>
<td>00010414</td>
<td>e49db004 pop {fp}</td>
</tr>
<tr>
<td>00010418</td>
<td>e12fffe bx lr</td>
</tr>
<tr>
<td>0001041c</td>
<td>000205c0 .word 0x000205c0</td>
</tr>
<tr>
<td>00010420</td>
<td>000205c4 .word 0x000205c4</td>
</tr>
<tr>
<td>00010424</td>
<td>000205c8 .word 0x000205c8</td>
</tr>
</tbody>
</table>

**ARM Machine Language**

Each machine language instruction is 32 bits

Fields within each machine language instruction identify:

- specific operation
- options
- source operands
- destination register
ARM Assembly Language

Express instructions in symbolic form, have software translate them into machine language instructions

Instructions: text file
Software: assembler

Assembler reads text file (".s" file), outputs to binary file (".o" file)

Summary: ARM microprocessor

CSE Pi array: 22 ARM V7 systems

ARM V7 characteristics

- RISC design
- Load-store architecture
- addresses: 32 bits
- address space: $2^{32} = 4$ GB
- register file: 16 32-bit registers (r0-r15)
- machine language instructions: 32 bits
ARM Integer Unit

Tightly coupled with Control Unit.

Handles all integer calculations.

Integer Unit Registers (32 bits):

- General purpose registers (r0 – r15)
- CPSR (current program status register):
  - includes NZCV status bits

ARM Data Manipulation
ARM Data Manipulation

- Data processing instructions (16):
  - bitwise (logical) instructions (4)
  - arithmetic (add/sub) instructions (6)
  - move instructions (2)
  - comparison (test) instructions (4)

Bitwise (logical) instructions

- Four instructions:
  - AND
  - ORR (inclusive or)
  - EOR (exclusive or)
  - BIC (bit clear)

- Three registers:
  result <= operand1 op operand2
Examples

- \( r_0 \leftarrow r_6 \text{ AND } r_7 \)
  
  \text{and } r_0, r_6, r_7
  
  e0060007

- \( r_1 \leftarrow r_6 \text{ OR } r_7 \)
  
  orr r1, r6, r7
  
  e1861007

Examples

- \( r_2 \leftarrow r_6 \text{ XOR } r_7 \)
  
  eor r2, r6, r7
  
  e0262007

- \( r_3 \leftarrow r_6 \text{ AND NOT } r_7 \)
  
  bic r3, r6, r7
  
  e1c63007
Decoding machine language

- assembly: `and r0, r6, r7`
- machine language: `e0060007`
- machine language in binary:
  
  \[
  \text{11100000000001100000000000000111}
  \]

Decoding machine language

- assembly: `orr r1, r6, r7`
- machine language: `e1861007`
- machine language in binary:
  
  \[
  \text{11100001100001100001000000000111}
  \]
Decoding machine language

- **assembly:** `eor r2, r6, r7`
- **machine language:** `e0262007`
- **machine language in binary:**

```
11100000001001100010000000000111
```

Decoding machine language

- **assembly:** `bic r3, r6, r7`
- **machine language:** `e1c63007`
- **machine language in binary:**

```
11100001110001100011000000000111
```
Arithmetic (add/sub) instructions

- Six instructions:
  - ADD
  - ADC (add with carry flag)
  - SUB
  - SBC (subtract with carry flag)
  - RSB (reverse subtract)
  - RSC (reverse subtract with carry flag)
- Three registers: result <= op1 op op2

Examples

- r0 <= r6 + r7
  add r0, r6, r7
  e0860007

- r1 <= r6 + r7 + carry flag
  adc r1, r6, r7
  e0a61007
Examples

- \( r_2 \leftarrow r_6 - r_7 \)
  
  \[
  \text{sub } r_2, r_6, r_7 \\
  \text{e0462007}
  \]

- \( r_3 \leftarrow r_6 - r_7 - \text{NOT carry flag} \)
  
  \[
  \text{sbc } r_3, r_6, r_7 \\
  \text{e0c63007}
  \]

Examples

- \( r_4 \leftarrow r_7 - r_6 \)
  
  \[
  \text{rsb } r_4, r_6, r_7 \\
  \text{e0664007}
  \]

- \( r_5 \leftarrow r_7 - r_6 - \text{NOT carry flag} \)
  
  \[
  \text{rsc } r_5, r_6, r_7 \\
  \text{e0e65007}
  \]
Decoding machine language

- assembly: `add r0, r6, r7`
- machine language: `e0860007`
- machine language in binary:
  \[
  11100000010001100000000000000111
  \]

Decoding machine language

- assembly: `sub r2, r6, r7`
- machine language: `e0462007`
- machine language in binary:
  \[
  11100000010001100010000000000111
  \]
Move instructions

- Two instructions:
  - MOV
  - MVN (move not)

- Two registers:
  result $$\leftarrow$$ operand2
  (operand1 is missing)

Examples

- r0 $$\leftarrow$$ r7
  
  ```
  mov r0, r7
  e1a00007
  ```

- r1 $$\leftarrow$$ NOT r7
  
  ```
  mvn r1, r7
  e1e01007
  ```
Decoding machine language

- assembly: `mov r0, r7`
- machine language: `e1a00007`
- machine language in binary:
  
  \[
  \begin{array}{cccccccccccccccc}
    1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
  \end{array}
  \]

  - `MOV` not used `r0` `r7`

Decoding machine language

- assembly: `mvn r1, r7`
- machine language: `e1e01007`
- machine language in binary:
  
  \[
  \begin{array}{cccccccccccccccc}
    1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
  \end{array}
  \]

  - `MVN` not used `r1` `r7`
Summary (so far)

- Each ARM machine language instruction is 32 bits wide (4 bytes)
  - Operation code: bits 24:21
  - Destination register: bits 15:12
  - Source register #1: bits 19:16
  - Source register #2: bits 3:0

- What are the other bits used for?

---

Data processing instructions

Format:

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>11-10</td>
<td>cond</td>
</tr>
<tr>
<td>9-8</td>
<td>0 0 I</td>
</tr>
<tr>
<td>7-0</td>
<td>opcode, S, Rn, Rd, shifter_operand</td>
</tr>
</tbody>
</table>

1 bit: Distinguishes between the immediate and register forms of shifter_operand.
S bit: Signifies that the instruction updates the condition codes.
Rn: Specifies the first source operand register.
Rd: Specifies the destination register.
shifter_operand: Specifies the second source operand. See Addressing Mode 1 - Data-processing operands on page A5-2 for details of the shifter operands.
Data processing instructions

- S bit: when the S bit is 1, NZCV condition code flags are updated by the instruction

- I bit: when I bit is 1, the second operand is an 8-bit immediate value (rather than a register)

- cond: will always be 1110 (for now)

Example: S bit

- \texttt{r10 <= r6 + r7}
  \begin{verbatim}
  add r10, r6, r7
  e086a007
  \end{verbatim}

- \texttt{r10 <= r6 + r7}, update NZCV flags
  \begin{verbatim}
  adds r10, r6, r7
  e096a007
  \end{verbatim}
Example: 1 bit

- $r0 \leftarrow r6 + 1$
  
  \[ \text{add } r0, r6, \#1 \]
  
  \[ e2860001 \]

- $r0 \leftarrow r6 + 0xff$
  
  \[ \text{add } r0, r6, \#0xff \]
  
  \[ e28600ff \]

Decoding machine language

- assembly: \texttt{add r0, r6, \#0xff}
- machine language: \texttt{e28600ff}
- machine language in binary:

```
11100010100001100000000011111111
```

I bit  S bit

\[ \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \uparrow \]

\[ \text{ADD} \quad r6 \quad r0 \quad 0xff \]
Example #12

Assembly language version of:

```c
int A = 12;
int B = 27;
int C = -1;

int main()
{
    C = A + B;
}
```

```asm
.global main
.text
main:  ldr   r0, =A          @ get address of A
       ldr   r1, [r0]        @ load value of A

       ldr   r2, =B          @ get address of B
       ldr   r3, [r2]        @ load value of B

       add   r4, r1, r3      @ sum = A + B

       ldr   r5, =C          @ get address of C
       str   r4, [r5]        @ store sum into C

exit:   bx   lr           @ return from main
```
Example #13

Demonstrates bitwise instructions

Based on code segment in Harris and Harris textbook (page 304)

Course website:

~cse320/Examples/example13.pdf

Example #14

Demonstrates shift instructions

Based on code segment in Harris and Harris textbook (page 305)

Course website:

~cse320/Examples/example14.pdf