Lecture Topics

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

Announcements

- Exam #1 results – will be sent via email
- Self-study Module #6
- Project #6 (due no later than 10/19)
Project #6

- Major themes:
  - programming in C (using "gcc")
  - integer bitwise operations
  - floating point arithmetic

- Library module (function "mult")

- Driver module (function "main")

Project #6

- 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 #6 (union)

union sp_union
{
    float frep;
    unsigned irep;
};

union sp_union num;
num.frep = 24.5;

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

Real: 24.500000 Hex: 41c40000

Project #6 (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, field1 \n", field1 ); field1: 00000089
printf( "field2: %08x, field2 \n", field2 ); field2: 000000ab
Project #6 (packing)

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

result = (value1 << 16) | value2;
```

```c
printf( "value1: %08x\n", value1 );  value1: 00001234
printf( "value2: %08x\n", value2 );  value2: 0000abcd
printf( "result: %08x\n", result );  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:
RAM[ PC ] → IR

Execute Phase:
decode IR
take appropriate action
update PC (usually PC + 4 → 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>r12: 000219d4</td>
<td>r13: 000219d8</td>
</tr>
<tr>
<td></td>
<td>r14: 000219dc</td>
<td>r15: 000103e8</td>
</tr>
</tbody>
</table>
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.
<1 lemon: > cat simple.c

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

int main()
{
    C = A + B;

    return 0;
}
```

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:

0001040c

Machine language instructions are 32 bits:

e1a00003

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

Addresses are 32 bits:

000205c0

Variables of type “int” are 32 bits:

0000000c
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

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000103e8</td>
<td>0000000c</td>
</tr>
<tr>
<td>000103ec</td>
<td>0000001b</td>
</tr>
<tr>
<td>000103f0</td>
<td>ffffffff</td>
</tr>
<tr>
<td>000103f4</td>
<td>0000000c</td>
</tr>
<tr>
<td>000103f8</td>
<td>0000001b</td>
</tr>
<tr>
<td>000103fc</td>
<td>0000000c</td>
</tr>
<tr>
<td>00010400</td>
<td>0000001b</td>
</tr>
<tr>
<td>00010404</td>
<td>ffffffff</td>
</tr>
<tr>
<td>00010408</td>
<td>0000000c</td>
</tr>
<tr>
<td>0001040c</td>
<td>0000000c</td>
</tr>
<tr>
<td>00010410</td>
<td>0000000c</td>
</tr>
<tr>
<td>00010414</td>
<td>0000000c</td>
</tr>
<tr>
<td>00010418</td>
<td>0000000c</td>
</tr>
<tr>
<td>0001041c</td>
<td>0000000c</td>
</tr>
<tr>
<td>00010420</td>
<td>0000000c</td>
</tr>
<tr>
<td>00010424</td>
<td>0000000c</td>
</tr>
</tbody>
</table>

Two segments:

0001xxxx: instructions
0002xxxx: data
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

- 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

- $r_0 <= r_6 \text{ AND } r_7$
  
  ```
  and r0, r6, r7
  e0060007
  ```

- $r_1 <= r_6 \text{ OR } r_7$
  
  ```
  orr r1, r6, r7
  e1861007
  ```

Examples

- $r_2 <= r_6 \text{ XOR } r_7$
  
  ```
  eor r2, r6, r7
  e0262007
  ```

- $r_3 <= r_6 \text{ AND NOT } r_7$
  
  ```
  bic r3, r6, r7
  e1c63007
  ```
Decoding machine language

- assembly: \texttt{and r0, r6, r7}
- machine language: \texttt{e0060007}
- machine language in binary:

\[
11100000000001100000000000000111
\]

Decoding machine language

- assembly: \texttt{orr r1, r6, r7}
- machine language: \texttt{e1861007}
- machine language in binary:

\[
11100001100001100001000000000111
\]
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

- \( r0 <= r6 + r7 \)
  
  \[
  \text{add} \quad r0, \ r6, \ r7 \\
  \text{e0860007}
  \]

- \( r1 <= r6 + r7 + \text{carry flag} \)
  
  \[
  \text{adc} \quad r1, \ r6, \ r7 \\
  \text{e0a61007}
  \]
Examples

- r2 <= r6 - r7
  
  \text{sub } r2, r6, r7
  
  e0462007

- r3 <= r6 - r7 - NOT carry flag
  
  \text{sbc } r3, r6, r7
  
  e0c63007

Examples

- r4 <= r7 - r6
  
  \text{rsb } r4, r6, r7
  
  e0664007

- r5 <= r7 - r6 - NOT carry flag
  
  \text{rsc } r5, r6, r7
  
  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:

```
11100000010000110000000000000111
```
Move instructions

- Two instructions:
  - MOV
  - MVN (move not)
- Two registers:
  result <= operand2
  (operand1 is missing)

Examples

- r0 <= r7
  `mov r0, r7`
  `e1a00007`
- r1 <= NOT r7
  `mvn r1, r7`
  `e1e01007`
Decoding machine language

- **assembly:** `mov r0, r7`
- **machine language:** `e1a00007`
- **machine language in binary:**
  
  \[
  11100001110100000000000000000111
  \]

Decoding machine language

- **assembly:** `mvn r1, r7`
- **machine language:** `e1e01007`
- **machine language in binary:**
  
  \[
  11100001111000000010000000000111
  \]
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></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>cond</td>
<td>0</td>
<td>0</td>
<td>I</td>
<td>opcode</td>
<td>S</td>
<td>Rn</td>
<td>Rd</td>
<td>shifter_operand</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

I 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

- `r10 <= r6 + r7`
  
  ```
  add r10, r6, r7
  e086a007
  ```

- `r10 <= r6 + r7`, update NZCV flags
  
  ```
  adds r10, r6, r7
  e096a007
  ```
Example: 1 bit

- r0 <= r6 + 1
  \[
  \text{add} \ r0, \ r6, \ #1 \\
  \text{e2860001}
  \]

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

Decoding machine language

- assembly: \text{add} \ r0, \ r6, \ #0xff
- machine language: e28600ff
- machine language in binary:
  \[
  \begin{array}{ccccccccccc}
  1110001 & 010000110 & 00000000 & 01111111 \\
  \text{ADD} & \text{r6} & \text{r0} & 0xff
  \end{array}
  \]

<table>
<thead>
<tr>
<th>I bit</th>
<th>S bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>