Lab2 : Debugging and Evaluation

Speaker: Chao-Chung Cheng
Directed by Prof. Tian-Sheuan Chang
March, 2004, NCTU
Goal of This Lab

- **Debug skills**
  - To debug both software running on processor and memory-mapped hardware design of the target platform.

- **Software cost estimation**
  - To estimate code sizes and benchmark performance

- **Profiling utility**
  - To estimate execution time of each function in an application

- **Memory configuration**
  - Performance/cost trade-off
Outline

- **Debugging Skills**
- **Software Quality Measurement (Evaluation)**
Introduction

Debug software in ARM platform

- Graphic user interface (GUI): ARM eXtended Debugger (AXD).
- Command line: ARM Symbolic Debugger (armsd).
Debug Target (1/2)

ARM Debugger
AXD

Remote Debug Interface (RDI)

Target (software)
ARMuulator
RDI
Target simulated in software

Target (hardware)
Multi-ICE
RDI
ARM development board

Angel
RDI
Remote_A
ARM development board
Multi-ICE connection
Basic Debug Requirements

- **Control of program execution**
  - set watchpoints on interesting data accesses
  - set breakpoints on interesting instructions
  - single step through code

- **Examine and change processor state**
  - view and set register values

- **Examine and change system state**
  - access system memory
    - download initial code

- **Interleaving source code**
  - show C/C++ code and disassembly code together
Register Banking

- **r0**
- **r1**
- **r2**
- **r3**
- **r4**
- **r5**
- **r6**
- **r7**
- **r8**
- **r9**
- **r10**
- **r11**
- **r12**
- **r13**
- **r14**
- **r15 (PC)**

- **usable in user mode**
- **system modes only**

- **CPSR**
- **SPSR_irq**
- **SPSR_svc**
- **SPSR_abt**

**Modes:**
- **user mode**
- **fiq mode**
- **svc mode**
- **abort mode**
- **irq mode**
- **undefined mode**

SOC Consortium Course Material
Outline

- Debugging skills
- *Software Quality Measurement*
Memory requirement of the program
- Memory type: volatile (RAM), non-volatile (ROM)
- Memory performance: access speed, data width, size and range

Profiling
- build up a picture of the percentage of time spent in each procedure.

Benchmarking performance
- Evaluate software performance prior to implement on hardware

Writing efficient C for ARM cores
- ARM/Thumb interworking
- Coding styles
**Application Code and Data Size**

`armlink` offers two options to provide the relevant information:

- **-info sizes** (sizes of all objects)
- **-info totals** (summary only)

<table>
<thead>
<tr>
<th></th>
<th>Code</th>
<th>RO Data</th>
<th>RW Data</th>
<th>ZI Data</th>
<th>Debug</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Object Totals</strong></td>
<td>25840</td>
<td>3444</td>
<td>0</td>
<td>0</td>
<td>104344</td>
</tr>
<tr>
<td><strong>Library Totals</strong></td>
<td>22680</td>
<td>762</td>
<td>0</td>
<td>300</td>
<td>9104</td>
</tr>
<tr>
<td><strong>Grand Totals</strong></td>
<td>48520</td>
<td>4206</td>
<td>0</td>
<td>300</td>
<td>113448</td>
</tr>
</tbody>
</table>

- **Total RO Size** (Code + RO Data) 52726 (51.49kB)
- **Total RW Size** (RW Data + ZI Data) 300 (0.29kB)
- **Total ROM Size** (Code + RO Data + RW Data) 52726 (51.49kB)

**The size of code/data in**
- an **ELF image** can be viewed using `fromelf -z`
- a **library** can be viewed using `armar -sizes`
Simple C routine
if (x>=0)
    return x;
else
    return -x;

The equivalent ARM assembly
labs CMP r0,#0 ;Compare r0 to zero
RSBLT r0,r0,#0 ;If r0<0 (less than=LT) then do r0= 0-r0
MOV pc,lr ;Move Link Register to PC (Return)

The equivalent Thumb assembly
CODE16 ;Directive specifying 16-bit (Thumb) instructions
labs CMP r0,#0 ;Compare r0 to zero
BGE return ;Jump to Return if greater or
 ;equal to zero
NEG r0,r0 ;If not, negate r0
return MOV pc,lr ;Move Link register to PC (Return)

<table>
<thead>
<tr>
<th>Code</th>
<th>Instructions</th>
<th>Size (Bytes)</th>
<th>Normalised</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM</td>
<td>3</td>
<td>12</td>
<td>1.0</td>
</tr>
<tr>
<td>Thumb</td>
<td>4</td>
<td>8</td>
<td>0.67</td>
</tr>
</tbody>
</table>
The linker calculates the ROM and RAM requirements for code and data as follows:

- **ROM**: Code size + RO data + RW data
- **RAM**: RW Data + ZI data.

You may wish to copy code from ROM into faster RAM, which will also increase the RAM requirements.

32-bit memory on-chip will significantly improve over 8 or 16-bit off-chip memory.
About Profiling:

– Profiler samples the **program counter** and computes the percentage time of each function spent.

– **Flat Profiling:**
  • If only pc-sampling info. is present. It can only display the time percentage spent in each function excluding the time in its children.
  • Flat profiling accumulates limited information without altering the image.

– **Call graph Profiling:**
  • If function call count info. is present. It can show the approximations of the time spent in each function including the time in its children.
  • Extra code is added to the image.
Profiling Limitations:

- Profiling is **NOT** available for code in ROM, or for scatter loaded images.
- No data is gathered for programs that are too small.
The Profiler command syntax is as follows:

```
```

### Call graph Profiling Sample Output

<table>
<thead>
<tr>
<th>Name</th>
<th>cum%</th>
<th>self%</th>
<th>desc%</th>
<th>calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>main</td>
<td>17.69%</td>
<td>60.06%</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>insert_sort</td>
<td>77.76%</td>
<td>17.69%</td>
<td>60.06%</td>
<td>1</td>
</tr>
<tr>
<td>strcmp</td>
<td>60.06%</td>
<td>0.00%</td>
<td></td>
<td>243432</td>
</tr>
<tr>
<td>qs_string_compare</td>
<td>3.21%</td>
<td>0.00%</td>
<td>13021</td>
<td></td>
</tr>
<tr>
<td>shell_sort</td>
<td>3.46%</td>
<td>0.00%</td>
<td>14059</td>
<td></td>
</tr>
<tr>
<td>insert_sort</td>
<td>60.06%</td>
<td>0.00%</td>
<td>243432</td>
<td></td>
</tr>
<tr>
<td>strcmp</td>
<td>66.75%</td>
<td>66.75%</td>
<td>0.00%</td>
<td>270512</td>
</tr>
</tbody>
</table>

### Note

- The `cum%` column represents the cumulative time spent on each function.
- The `self%` column represents the self-time spent on each function.
- The `desc%` column represents the time spent by the function's descendants.
Execution time (real-time vs. emulated)

- $sys\_clock$
- Execution time = Total Cycle count * Cycle time

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$instructions[Total]</td>
<td>152136792</td>
</tr>
<tr>
<td>_Core_Cycles</td>
<td>291463836</td>
</tr>
<tr>
<td>_S_Cycles</td>
<td>180157645</td>
</tr>
<tr>
<td>_N_Cycles</td>
<td>85279320</td>
</tr>
<tr>
<td>_I_Cycles</td>
<td>26827012</td>
</tr>
<tr>
<td>_C_Cycles</td>
<td>0</td>
</tr>
<tr>
<td>_Total</td>
<td>292263977</td>
</tr>
</tbody>
</table>

```
$rdi_log
$target_fpu
$image_cache_enable
$clock
$ARM7TDMI$irq
$ARM7TDMI$fiq
$ARM7TDMI$config
$ARM7TDMI$acmd
$ARM7TDMI$cputime
$ARM7TDMI$sys_clock
```

0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000 0x00000000

292263977 292263977
When ARM processor executes program, it will change these clock types according to the demand of operating.

Cycle Types (Von Neuman Cores) for ARM7TDMI

- **N-cycles (Non-sequential cycle)**
  - The ARM core requests a transfer to or from an address which is unrelated to the address used in the preceding cycle.

- **S-cycles (Sequential cycle)**
  - The ARM core requests a transfer to or from an address which is either the same, or one word or one-half-word greater than the preceding address.

- **I-cycles (Internal cycle)**
  - The ARM core does not require a transfer, as it is performing an internal function.

- **C-cycles (Coprocessor register transfer cycle)**
  - The ARM core wishes to use the data bus to communicate with a coprocessor, but does not require any action by the memory system.

- **Total clock cycle = (N + S + I + C + Waits)-cycles**
## Cycle Types (Harvard Cores) for ARM9TDMI

<table>
<thead>
<tr>
<th>Cycle types</th>
<th>Instruction bus</th>
<th>Data bus</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core-cycles</td>
<td>-</td>
<td>-</td>
<td>The total number of core clock. (including pipeline stalls due to interlocks and multiple cycle instructions)</td>
</tr>
<tr>
<td>ID-cycles</td>
<td>Active</td>
<td>Active</td>
<td>-</td>
</tr>
<tr>
<td>I-cycles</td>
<td>Active</td>
<td>Idle</td>
<td>-</td>
</tr>
<tr>
<td>D-cycles</td>
<td>Idle</td>
<td>Active</td>
<td>-</td>
</tr>
<tr>
<td>Idle-cycles</td>
<td>Idle</td>
<td>Idle</td>
<td>-</td>
</tr>
</tbody>
</table>

- Total clock cycle = (Core + ID + I + D + idle + Waits)-cycles
Estimation using different Memory model

- If no map file is specified:
  - ARMulator will use a 4GB bank of ‘ideal’ memory, i.e., no wait states.

- The map file defines regions of memory, and, for each region:
  - The address range
  - The data bus width (in bytes).
  - The access times (in ns)

armsd.map typically contains memory map information:

<table>
<thead>
<tr>
<th>start address</th>
<th>length</th>
<th>label</th>
<th>width</th>
<th>access</th>
<th>read time</th>
<th>write time</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>00020000</td>
<td>ROM</td>
<td>2</td>
<td>R</td>
<td>150/100</td>
<td>150/100</td>
</tr>
<tr>
<td>10000000</td>
<td>00008000</td>
<td>RAM</td>
<td>4</td>
<td>RW</td>
<td>100/65</td>
<td>100/65</td>
</tr>
</tbody>
</table>
Benchmarking cached cores

- Cache efficiency
  - Avg. memory access time = hit time + Miss rate * Miss Penalty
  - Cache Efficiency = Core-Cycles / Total Bus Cycles

![Image of debugging interface]
Efficient Code Development

- ARM/Thumb interworking
  - ARM: bottleneck, interrupt handle
  - Thumb: code size

- Compiler optimization:
  - Space or speed (e.g., -Ospace(default) or -Otime)
  - Effort (e.g., -O0, -O1 or -O2)
  - Instruction scheduling

- Coding style
  - Variable type and size
  - Parameter passing
  - Loop termination
  - Division operation and modulo arithmetic
Data Layout

Default
char a;
short b;
char c;int d;

<table>
<thead>
<tr>
<th>a</th>
<th>pad</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>pad</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>

occupies 12 bytes, with 4 bytes of padding

Optimized
char a;
char c;
short b;int d;

<table>
<thead>
<tr>
<th>a</th>
<th>c</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>d</td>
</tr>
</tbody>
</table>

occupies 8 bytes, without any padding

• Group variables of the same type together
  • The best way to ensure that as little padding data as possible is added by the compiler.
Variable Types – Size Examples

int intinc
(int a)
{ return a + 1;
}
short shortinc
(short a)
{ return a + 1;
}
char charinc (char a)
{ return a + 1;
}
Stack Usage

Stack usage:
- Return addresses for subroutines
- Local arrays & structures

To minimize stack usage:
- Keep functions small (few variables, less spills)
- Minimize the number of ‘live’ variables (i.e., those which contain useful data at each point in the function)
- Avoid using large local structures or arrays (use dynamic allocation malloc/free instead)
- Avoid recursion
Global Data Issues

- How much **space the variables occupy at run time**.
  - This determines the **size of RAM** required for a program to run. The ARM compilers may insert padding bytes between variables, to ensure that they are properly aligned.

- How much **space the variables occupy in the image**.
  - This is one of the factors determining the **size of ROM** needed to hold a program. Some global variables which are not explicitly initialized in your program may nevertheless have their initial value (of zero, as defined by the C standard) stored in the image.

- How many **codes need to access the variables**.
  - Some data organizations require more code to access the data. As an extreme example, the smallest data size would be achieved if all variables were stored in suitably sized bitfields, but the code required to access them would be much larger (trade-off between size and performance)
Loop termination

```c
... int acc(int n) {
    int i;       //loop index
    int sum=0;

    for (i=1; i<=n ;i++)
        sum+=i;
    return sum;
}
... loop.c
```

```c
... int acc(int n) {
    int i;       //loop index
    int sum=0;

    for (i=n; i!=0 ;i--)
        sum+=i;
    return sum;
}
... loop_opt.c
```
The remainder operator ‘%’ is commonly used in modulo arithmetic.

- This will be expensive if the modulo value is **not a power of two**.
- This can be avoided by rewriting C code to use `if ()` statement heck.

```c
unsigned counter1 (unsigned counter)
{ return (++counter % 60);
}
```

Counter1

```c
STMFB sp!, {lr}
ADD r1, r0, #1
MOV r0, #0x3C
BL __rt_udiv
MOV r0, r1
LDMIA sp!, {pc}
```

```c
unsigned counter2 (unsigned counter)
{ if (++counter >= 60)
    counter=0;
    return counter
}
```

Counter2

```c
ADD r0, r0, #1
CMP r0, #0x3C
MOVCS r0, #0
MOV pc, lr
```

```c
modulo.c
```

```c
modulo_opt.c
```
ARM Processor Pipeline Operations

ARM7TDMI
- Fetch
  - instruction fetch
- Decode
  - Thumb decompress
  - ARM decode
- Execute
  - reg read
  - shift/ALU
  - reg write

ARM9TDMI
- Fetch
  - instruction fetch
- Decode
  - r read decode
- Execute
  - shift/ALU
- Memory
  - data memory access
- Write
  - reg write

ARM10TDMI
- Fetch
- Decode
- Execute
  - addr calc.
  - data memory access
- Memory
  - data write
- Write
  - reg write
Dhrystone Result Example

Target: ARM940T, 4kB I-cache, 4kB D-cache, 10.00MHz core clock, (Physical memory, 3.3MHz)

<table>
<thead>
<tr>
<th></th>
<th>Instructions</th>
<th>Core Cycles</th>
<th>S-cycles</th>
<th>N-cycles</th>
<th>I-cycles</th>
<th>C-cycles</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iteration 1</td>
<td>306</td>
<td>446</td>
<td>377</td>
<td>0</td>
<td>345</td>
<td>0</td>
<td>722</td>
</tr>
<tr>
<td>Iteration n</td>
<td>306</td>
<td>446</td>
<td>7</td>
<td>0</td>
<td>142</td>
<td>0</td>
<td>149</td>
</tr>
</tbody>
</table>

Iteration 1: \(673 \times 1 / 3,333,333 = 216.6\text{us}\)
Iteration n: \(149 \times 1 / 3,333,333 = 44.7\text{us}\)
\(446/149 = 2.993\)

Iteration 1~n: Total Core Cycles: 27074407
Total Bus Cycles : 9034428
Cache Efficiency : 2.9979 (MCCFG=3)
Cache Efficiency % : 100 x (Cache Efficiency x MCCFG) = 99.93%

Cached with different clock domains
Dhrystone Analysis

Cached with different clock domains

<table>
<thead>
<tr>
<th></th>
<th>ARM9TDMI 10MHz Mem</th>
<th>AR940T 10MHz Core, 3.3MHz Mem, Cache On</th>
<th>AR940T 10MHz Core, 3.3MHz Mem, Cache Off</th>
<th>ARM9TDMI 3.3MHz Mem</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Performance</td>
<td>100</td>
<td>100</td>
<td>5.64</td>
<td>33.3</td>
</tr>
</tbody>
</table>

SOC Consortium Course Material
Lab 2: Debugging and Evaluation

Goal
- How to perform a variety of debugging tasks and software quality evaluation
- How to profile and evaluation the software performance of the application

Guidance
- Instruction of this lab
- Preconfigured project stationery files

Steps
- Debugging skills
- Software Quality Measurement

Requirements and Exercises
- See note file

Discussion
- The approaches to minimize the code size and enhance the performance of the program
- The advantages of ARM/Thumb instruction sets interworking