Debugging and Evaluation

Speaker: Juin-Nan Liu

Adopted from National Chiao-Tung University
SOC Course Material

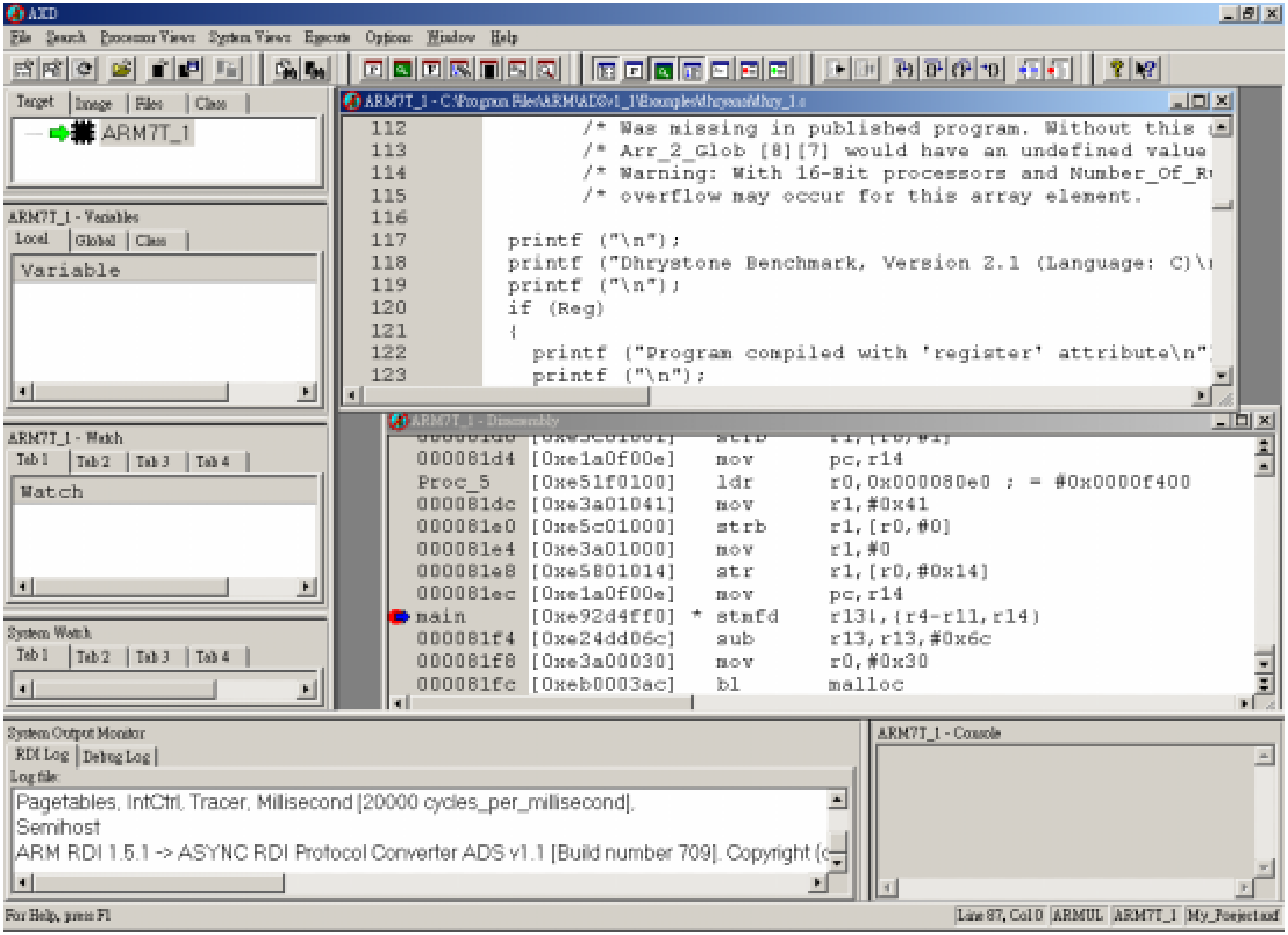
Goal of This Lab

- ❑ Debug skills to be used to debug both software of processor and memory-mapped hardware design running at the target platform
- ❑ Software cost estimation
  - – The cost of a program includes Read Only (RO) data, Read Write (RW) data and Zero-Initialized (ZI) data
- ❑ Profiling utility
  - – Can be used to estimate percentage time of each function in an application
- ❑ Memory configuration
  - – For performance/cost trade-off
  - – E.g., an embedded system might use fast, 32-bit RAM for performance-critical code, such as interrupt handlers and the stack, slower 16-bit RAM for application RW data, and ROM for normal application code

Outline

- ***Debugging skills***
- Software Quality Measurement

AXD Desktop



```
        /* Was missing in published program. Without this (
        /* Arr_2_Glob [8][7] would have an undefined value
        /* Warning: With 16-Bit processors and Number_Of_R
        /* overflow may occur for this array element.

printf ("\n");
printf ("Dhrystone Benchmark, Version 2.1 (Language: C)\n
printf ("\n");
if (Reg)
{
  printf ("Program compiled with 'register' attribute\n")
  printf ("\n");
```

```
000081d4 [0xe1a0f00e]   mov    pc,r14
Proc_5   [0xe51f0100]   ldr    r0,0x000080e0 ; = #0x0000f400
000081dc [0xe3a01041]   mov    r1,#0x41
000081e0 [0xe5c01000]   strb   r1,[r0,#0]
000081e4 [0xe3a01000]   mov    r1,#0
000081e8 [0xe5801014]   str    r1,[r0,#0x14]
000081ec [0xe1a0f00e]   mov    pc,r14
main     [0xe92d4ff0] * stmfd  r13!,{r4-r11,r14}
000081f4 [0xe24dd06c]   sub    r13,r13,#0x6c
000081f8 [0xe3a00030]   mov    r0,#0x30
000081fc [0xeb0003ac]   bl     malloc
```

```
Pagetables, IntCtrl, Tracer, Millisecond [20000 cycles_per_millisecond],
Semihost
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter ADS v1.1 [Build number 709], Copyright (c
```

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
  - read and write register values
- Examine and change system state
  - access to system memory
    - download initial code

Outline

- ❑ Debugging skills
- ❑ ***Software Quality Measurement***

Software Quality Measurement (1/2)

- Memory requirement of the program
  - Data 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.
- Evaluate software performance prior to implement on hardware
- Writing efficient C for ARM cores
  - ARM/Thumb interworking
  - Coding styles

Software Quality Measurement (2/2)

- Performance Benchmarking
  - Harvard Core
    - D-cycles, ID-cycles, I-cycles
  - von Newman Cores
    - N-cycles, S-cycles, I-Cycles, C-Cycles
  - Clock rate
    - Processor, external bus
  - Cache efficiency
    - Average memory access time = hit time +Miss rate x Miss Penalty
    - Cache Efficiency = Core-Cycles / Total Bus Cycles

Application Code and Data Size

- armlink offers two options to provide the relevant information:
  - -info sizes (sizes of all objects)
  - -info totals (summary only)

```
==========================================================
Image component sizes
   Code    RO Data    RW Data    ZI Data      Debug
  25840      3444          0          0     104344   Object Totals
  22680       762          0        300       9104   Library Totals
==========================================================
   Code    RO Data    RW Data    ZI Data      Debug
  48520      4206          0        300     113448   Grand Totals
==========================================================
  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

ARM and Thumb Code Size

Simple C routine

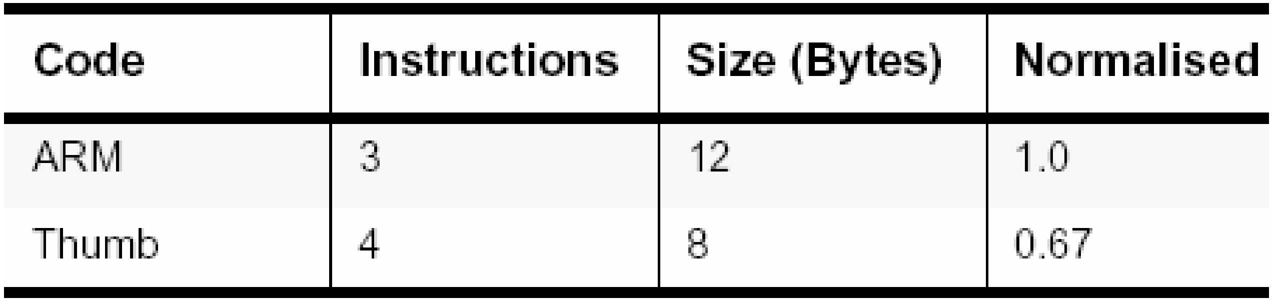
```
if (x>=0)
    return x;
else
    return -x;
```

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-r
        MOV     pc,lr       ;Move Link Register to PC (Return)
```

The equivalent Thumb assembly

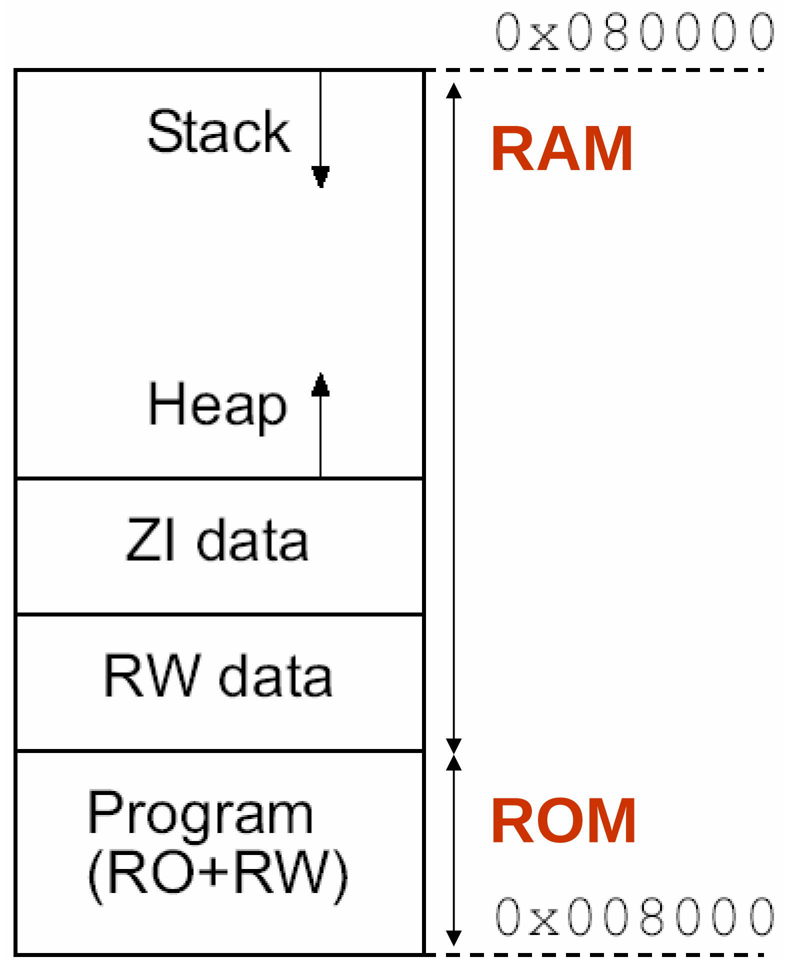
```
        CODE16 ;Directive specifying 16-bit (Thumb) instruction
labs    CMP     r0,#0       ;Compare r0 to zero
        BGE     return      ;Jump to Return if greater or
                            ;equal to zero
        NEG     r0,r0       ;If not, negate r0
return  MOV     pc,lr       ;Move Link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



Code	Instructions	Size (Bytes)	Normalised
ARM	3	12	1.0
Thumb	4	8	0.67

Memory Map and Size Considerations

- 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
- Placing the **stacks** in zero-wait state, 32-bit memory on-chip will significantly improve over -8 or 16- bit off-chip memory



Default memory map

ARM Profiler

- 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
- Limitations:
  - Profiling is **NOT** available for code in ROM, or for scatter loaded images.
  - No data is gathered for programs that are too small.

Profiler Command-line Options

- The command syntax is as follows:

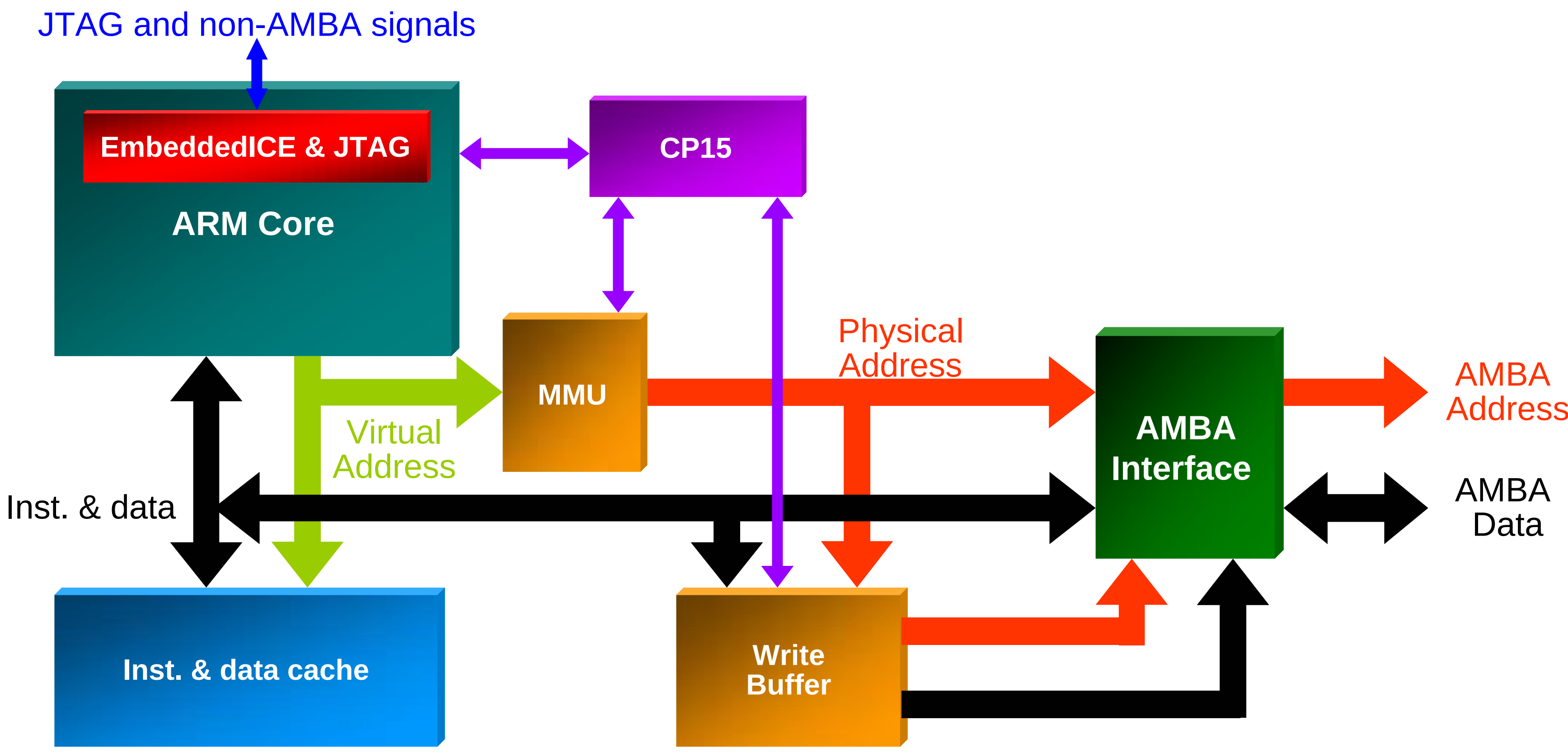
  armprof [-parent|-noparent] [-child|-nochild] [-sort options] prf_file

  cumulative
  self
  descendants
  calls

- Sample Output

```
Name                     cum%    self%    desc%   calls
-----------------------------------------------------------------
 main                            17.69%   60.06%  1
insert_sort      77.76%  17.69%  60.06%   1
 strcmp                  60.06%  0.00%    243432
-----------------------------------------------------------------
 qs_string_compare       3.21%   0.00%    13021
 shell_sort              3.46%   0.00%    14059
 insert_sort             60.06%  0.00%    243432
strcmp           66.75%  66.75%  0.00%    270512
-----------------------------------------------------------------
```

In ARM Macrocell



Cycle Types, Von Neuman Cores

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, and no useful prefetching can be performed at the same time.
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	The sum of the S-Cycles, N-Cycles, I-Cycles and C-Cycles.

Map File

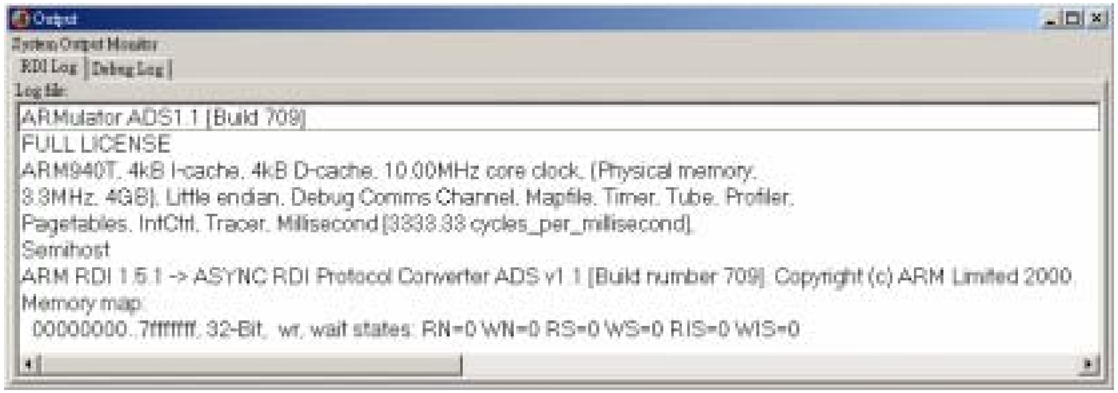
- If no map file is specified:
  - ARMulator will use a 4GB bank of ‘ideal’ memory, i.e., no wait states.
- The map file defines regions of memory, and, for each region:
  - The address range to which that region is mapped.
  - The data bus width (in bytes).
  - The access times for the memory region (in ns)
- armsd.map typically contains something like:

```
00000000  00020000  ROM  2  R    150/100  150/100
10000000  00008000  RAM  4  RW   100/65   100/65
```

  - Columns are (left to right):

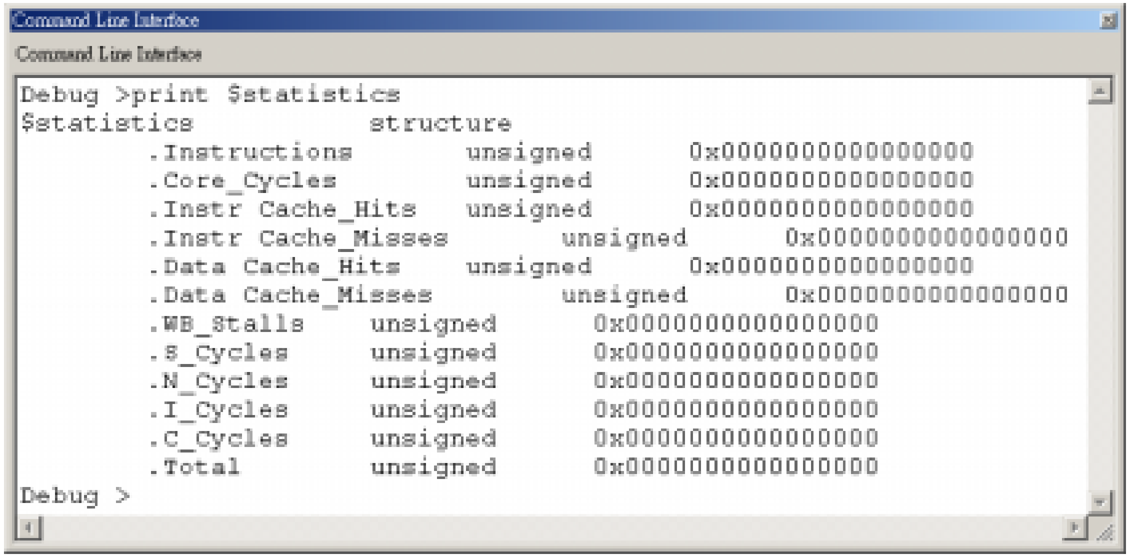
    start address (in hex)
    length (in hex)
    name
    width (1, 2, or 4 bytes)
    access type (read-only or read/write)
    read timing in ns (NON-Seq / Seq)
    writing timing in ns (NON-Seq / Seq)

Configure for Target System



```
ARMulator ADS1.1 [Build 709]
FULL LICENSE
ARM940T, 4kB I-cache, 4kB D-cache, 10.00MHz core clock, (Physical memory,
3.3MHz, 4GB), Little endian, Debug Comms Channel, Mapfile, Timer, Tube, Profiler,
Pagetables, IntCtrl, Tracer, Millisecond [3333.33 cycles_per_millisecond],
Semihost
ARM RDI 1.5.1 -> ASYNC RDI Protocol Converter ADS v1.1 [Build number 709], Copyright (c) ARM Limited 2000
Memory map:
 00000000..7fffffff, 32-Bit,  wr, wait states: RN=0 WN=0 RS=0 WS=0 RIS=0 WIS=0
```

ARMulator startup Message



```
Debug >print $statistics
$statistics           structure
        .Instructions       unsigned      0x0000000000000000
        .Core_Cycles        unsigned      0x0000000000000000
        .Instr Cache_Hits   unsigned      0x0000000000000000
        .Instr Cache_Misses       unsigned      0x0000000000000000
        .Data Cache_Hits    unsigned      0x0000000000000000
        .Data Cache_Misses        unsigned      0x0000000000000000
        .WB_Stalls    unsigned      0x0000000000000000
        .S_Cycles     unsigned      0x0000000000000000
        .N_Cycles     unsigned      0x0000000000000000
        .I_Cycles     unsigned      0x0000000000000000
        .C_Cycles     unsigned      0x0000000000000000
        .Total        unsigned      0x0000000000000000
Debug >
```

Cached core additional statistics

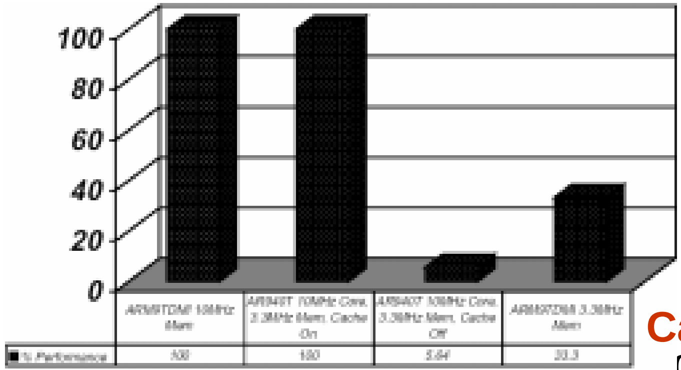
Dhrystone Result Example

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

	Instructions	Core Cycles	S-cycles	N-cycles	I-cycles	C-cycles	Total
Iteration 1	306	446	377	0	345	0	722
Iteration n	306	446	7	0	142	0	149

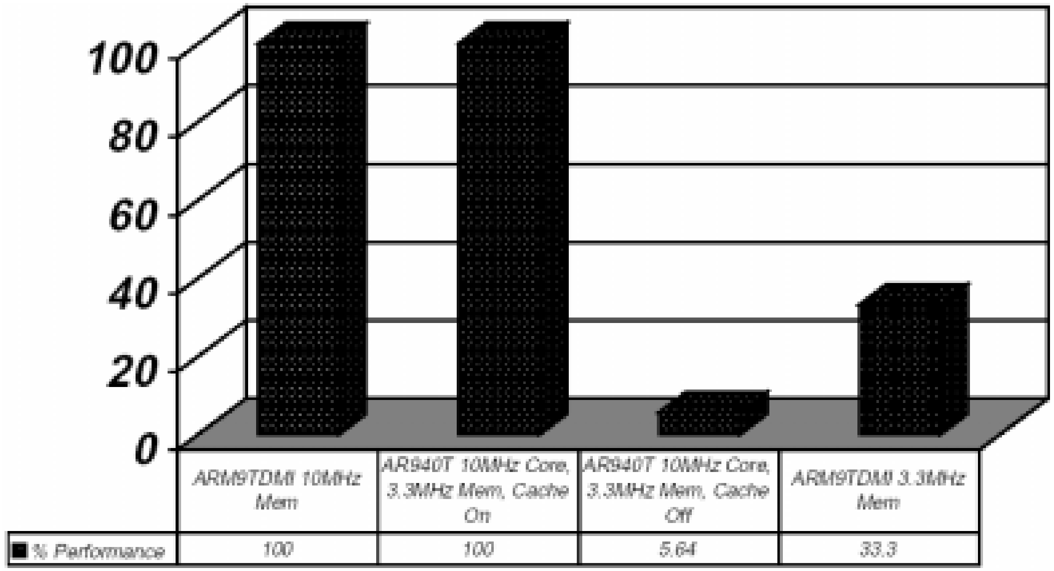
Iteration 1: 673 x 1 / 3,333,333 = 216.6us
Iteration n: 149 x 1 / 3,333,333 = 44.7us
446/149 = 2.993

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%



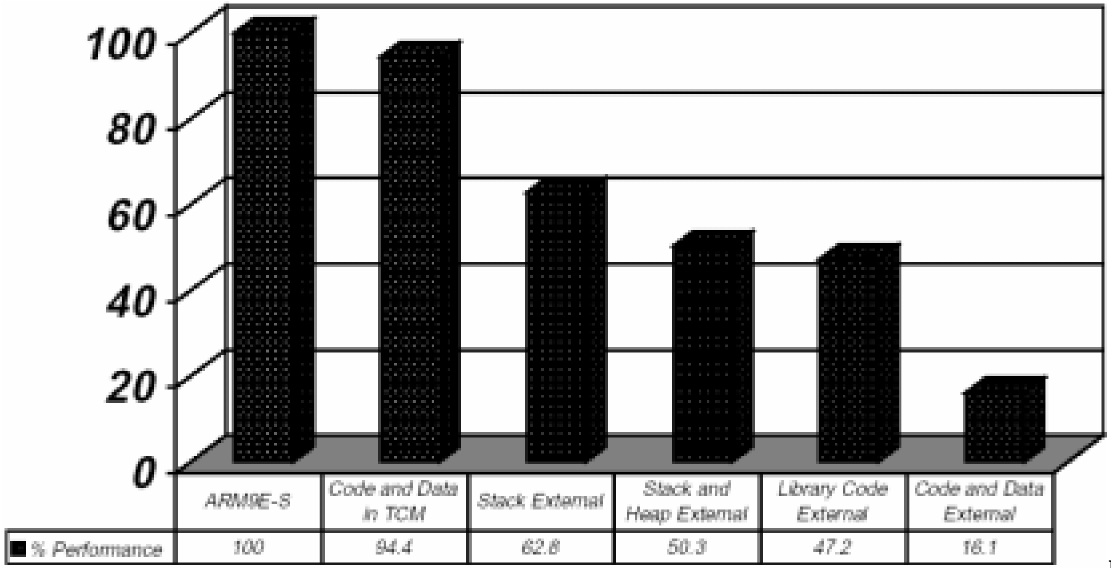
Cached with different clock domains

Dhrystone Analysis



	ARM9TDMI 10MHz Mem	AR940T 10MHz Core, 3.3MHz Mem, Cache On	AR940T 10MHz Core, 3.3MHz Mem, Cache Off	ARM9TDMI 3.3MHz Mem
■ % Performance	100	100	5.64	33.3

Cached with different clock domains



	ARM9E-S	Code and Data in TCM	Stack External	Stack and Heap External	Library Code External	Code and Data External
■ % Performance	100	94.4	62.8	50.3	47.2	16.1

TCM on ARM966E-S

Code Development

- General/Machine-dependent guideline
  - Compiler optimization:
    - Space or speed (e.g, **-Ospace** or **-Otime**)
    - Debug or release version (e.g., **-O0** ,**-O1** or **-O2**)
    - Instruction scheduling
  - Coding style
    - Parameter passing
    - Loop termination
    - Division operation and modulo arithmetic
    - Variable type and size

Remainders – Modulo Arithmetic

- 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 avoid by rewriting C code to use **if ()** statement heck

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


counter1
    STMFB   sp!, {lr}
    ADD     r1, r0, #1
    MOV     r0, #0x3C
    BL      __rt_udiv
    MOV     r0, r1
    LDMIA   sp!, {pc}
```

```
unsigned counter2 (unsigned
counter)
{ if (++counter >= 60)
        counter=0;
    return counter
}
counter2
    ADD     r0, r0, #1
    CMP     r0, #0x3C
    MOVCS   r0, #0
    MOV     pc, lr
```

Variable Types – Size Examples

```
int wordinc
(int a)
{ return a + 1;
}
```

```
wordinc
ADD a1,a1,#1
MOV pc,lr
```

```
short shortinc
(short a)
{ return a + 1;
}
```

```
shortinc
ADD a1,a1,#1
MOV a1,a1,LSL #16
MOV a1,a1,ASR #16
MOV pc,lr
```

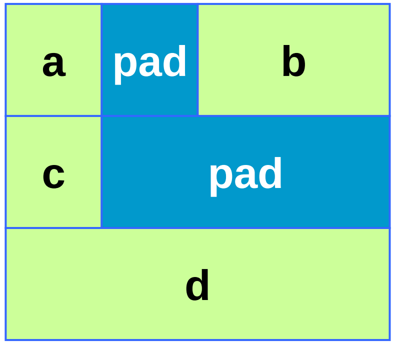
```
char charinc (char
a)
{ return a + 1;
}
```

```
charinc
ADD a1,a1,#1
AND a1,a1,#&ff
MOV pc,lr
```

Data Layout

Default

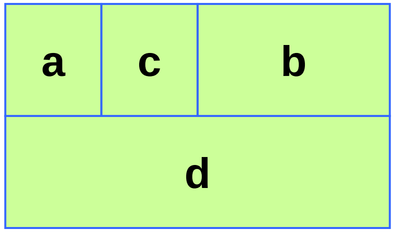
```
char a;
short b;
char c;
int d;
```



occupies 12 bytes, with 4 bytes of padding

Optimized

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



occupies 8 bytes, without any padding

Group variables of the same type together. This is the best way to ensure that as little padding data as possible is added by the compiler.

Stack Usage

- C/C++ code uses the stack intensively. The stack is used to hold:
  - 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 malloc/free instead)
  - Avoid recursion

Global Data Issues

- ❑ When declaring global variables in source code to be compiled with ARM Software, three things are affected by the way you structure your code:
  - 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.
  - The **size of the code needed 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.