Code Development

Speaker: Fury Chao-Chung Cheng
Directed by Prof. Tian-Sheuan Chang

March, 2004

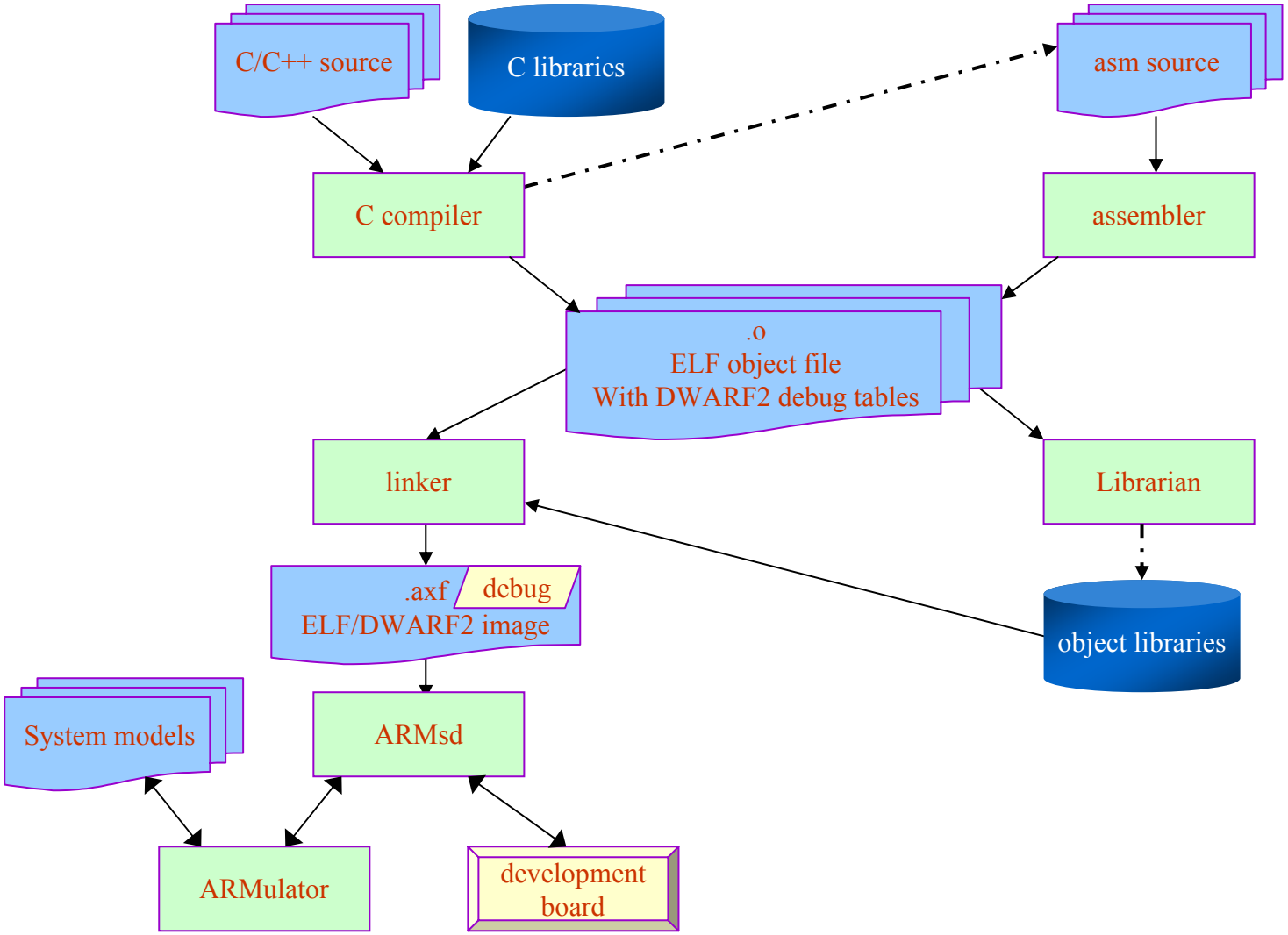
Goal of This Lab

- Familiarize with ARM software development tools: ARM Development Suite (ADS)
  - Project management
  - Configuring the settings of build targets for your project
- Writing code for ARM-based platform design
- Mixed instruction sets, ARM and Thumb interworking, is learned to balance the performance and code density of an application.

Outline

- *Basic Code Development*
- ARM/Thumb Interworking
- ARM/Thumb Procedure Call Standard
- Lab1 – Code Development

The Structure of ARM Tools



DWARF: Debug With Arbitrary Record Format

ELF: Executable and linking format

Main Components in ADS (1/2)

- ANSI C compilers – armcc and tcc
- ISO/Embedded C++ compilers – armcpp and tcpp
- ARM/Thumb assembler - armasm
- Linker - armlink
- Project management tool for windows - CodeWarrior
- Instruction set simulator - ARMulator
- Debuggers - AXD, ADW, ADU and armsd
- Format converter - fromelf
- Librarian – armar
- ARM profiler - armprof

ADS: ARM Developer Suite

Main Components in ADS (2/2)

- C and C++ libraries
- ROM-based debug tools (ARM Firmware Suite, AFS)
- Real Time Debug and Trace support
- Support for all ARM cores and processors including ARM9E, ARM10, Jazelle, StrongARM and Intel Xscale

View in CodeWarrier

- The CodeWarrior IDE provides a simple, versatile, graphical user interface for managing your software development projects.
- Develop C, C++, and ARM assembly language code
- targeted at ARM and Thumb processors.
- It speeds up your build cycle by providing:
  - comprehensive project management capabilities
  - code navigation routines to help you locate routines quickly.

CodeWarrior Desktop

Menu

Toolbar

Editor windows

Project Files view

Target Settings

Metrowerks CodeWarrior for ARM Developer Suite v1.2

File Edit View Search Project Debug Window Help

lencod.cc

Path: D:\fury\AVC_ARM\A...\lencod.cc

```
void Clear_Motion_Search_Module ();
int test_fury;
int main(int argc,char **argv)
{
  test_fury=999;
  p_dec = p_stat = p_log = p_trace = NULL;

  Configure (argc, argv);
  init_img();
  AllocNalPayloadBuffer();

  frame_pic = malloc_picture();
  if (input->InterlaceCodingOption != FRAME_
  {
    top_pic = malloc_picture();
```

Line 270 Col 31

AVC_ARM.mcp

DebugRel

Files | Link Order | Targets

File	Code	Data
decoder.cc	6580	24
filehandle.cc	424	4
fmo.cc	4132	88
header.cc	5136	160
image.cc	35532	417
intrarefresh.cc	492	20
leaky_bucket.cc	1804	40252
lencod.cc	25356	82142
loopFilter.cc	3656	480
macroblock.cc	31572	560
mb_access.cc	3180	0
mbuffer.cc	14528	493
66 files	272K	163K

DebugRel Settings

Target Settings Panels

- Target
  - Target Settings
  - Access Paths
  - Build Extras
  - Runtime Settings
  - File Mappings
  - Source Trees
  - ARM Target
- Language Settings
  - ARM Assembler
  - ARM C Compiler
  - ARM C++ Compiler
  - Thumb C Compiler
  - Thumb C++ Compi...
- Linker
  - ARM Linker
  - ARM fromELF
- Editor

Target Settings

Target Name: DebugRel

Linker: ARM Linker

Pre-linker: None

Post-linker: None

Output Directory: {Project} Choose Clear

Save project entries using relative paths

Factory Settings | Revert | Import Panel... | Export Panel...

OK | Cancel | Apply

Views in AXD

- Various views allow you to **examine** and **control** the processes you are debugging.
- In the main menu bar, two menus contain items that display views:
  - The items in the **Processor Views menu** display views that apply to the **current processor only**
  - The items in the **System Views menu** display views that apply to the entire, possibly **multiprocessor**, target system

AXD: the ARM eXtended Debugger

AXD Desktop

Menu

Toolbar

Watch system view

Variable processor view

Watch processor view

Control System view

Source processor view

Disassembly processor view

Console processor view

Control System view

Status bar

ARMulator (1/2)

- A suite of programs that models the behavior of various ARM processor cores and system architecture in software on a host system
- Can be operated at various levels of accuracy
  - Instruction accurate
  - Cycle accurate
  - Timing accurate

ARMulator (2/2)

- Benchmarking before hardware is available
  - Instruction count or number of cycles can be measured for a program.
  - Performance analysis.
- Run software on ARMulator
  - Through ARMsd or ARM GUI debuggers, e.g., AXD
  - The processor core model incorporates the remote debug interface, so the processor and the system state are visible from the ARM symbolic debugger
  - Supports a C library to allow complete C programs to run on the simulated system

ARM Symbolic Debugger

- ARMsd: ARM and Thumb symbolic debugger
  - can single-step through C/C++ or assembly language sources
  - set break-points and watch-points
  - examine program variables or memory
- It is a front-end interface to debug program running either
  - under simulation (on the ARMulator)
  - remotely on a ARM development board (via a serial line or through JTAG test interface)
- It allows the setting of
  - breakpoints, addresses in the code
  - watchpoints, memory address if accessed as data address

  ➔ cause exception to halt so that the processor state can be examined

Basic Debug Requirements

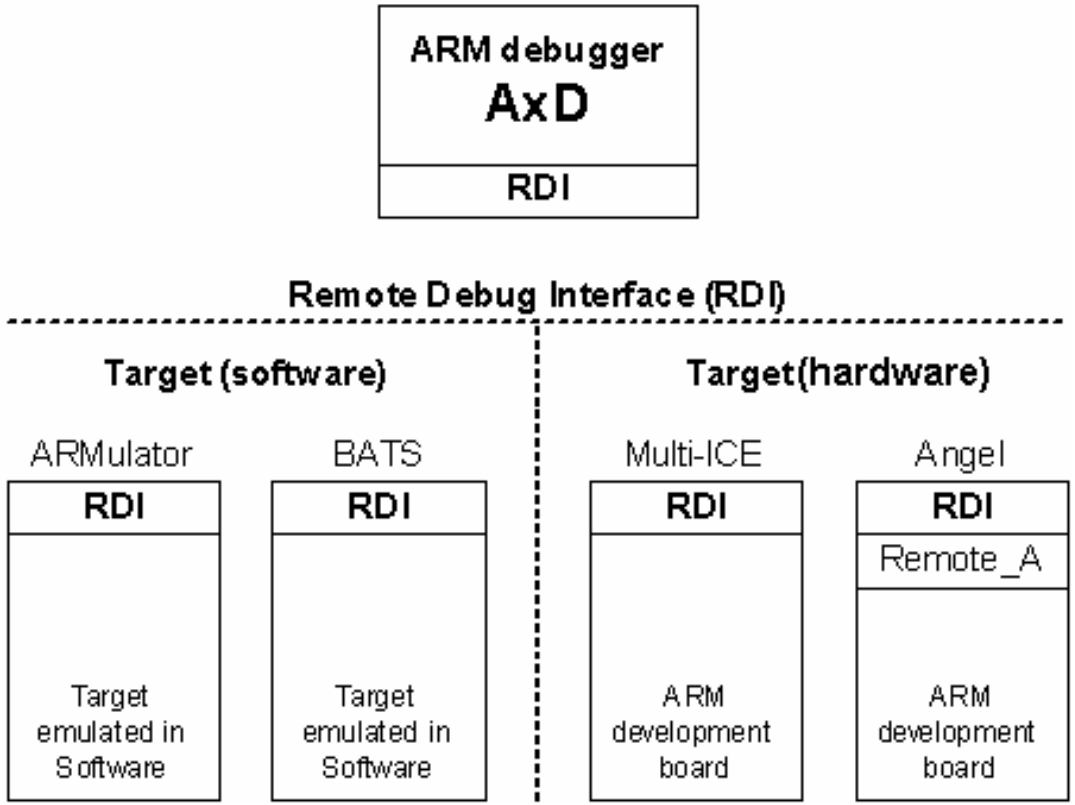
- Control of program execution
  - set watchpoints on interesting data accesses
  - set breakpoints on interesting instructions
  - single step through code
- Examine and change processor state
  - read and write register values
- Examine and change system state
  - access to system memory
    - download initial code

Debugger (1/2)

- Debugger
  - to examine and control the execution of software
    - setting breakpoints/watchpoints
    - reading from / writing to memory
  - Different forms of the debug target
    - software (early stage of product development)
    - Prototype (on a PCB including one or more processors)
    - final product
  - The debugger issues instructions
    - load software into memory on the target
    - start and stop execution of that software
    - display the contents of memory, registers, and variables
    - change stored values

Debugger (2/2)

- Debug agents
  - Multi-ICE
  - Embedded ICE
  - ARMulator
  - BATS
  - Angel
- Remote Debug Interface (RDI)
  - an open ARM standard procedural interface between a debugger and a debug agent



Program Design

- Start with understanding the requirements, translate the requirements into an unambiguous specifications
- Define a program structure, the data structure and the algorithms that are used to perform the required operations on the data
  - The algorithms may be expressed in pseudo-code
- Individual modules should be coded, tested and documented
  - Nearly all programming is based on high-level languages, however it may be necessary to develop small software components in assembly language to get the best performance

Outline

- Basic Code Development
- ***ARM/Thumb Interworking***
- ARM/Thumb Procedure Call Standard
- Lab1 – Code Development

ARM Instruction Sets

- ARM processor is a 32-bit architecture, most ARM processors implement two instruction sets
  - 32-bit **ARM** instruction set
  - 16-bit **Thumb** instruction set

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-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)
```

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

The Need for Interworking

- The code density of Thumb and its performance from small memory sizes make it ideal for the bulk of C code in many systems. However there is still a need to change between ARM and Thumb state within most applications:
  - ARM code provides better performance from wide memory
    - therefore ideal for speed-critical parts of an application
  - some functions can only be performed with ARM instructions, e.g.
    - access to CPSR (to enable/disable interrupts & to change mode)
    - access to coprocessors
  - exception Handling
    - ARM state is automatically entered for exception handling, but system specification may require usage of Thumb code for main handler
  - simple standalone Thumb programs will also need an ARM assembler header to change state and call the Thumb routine

Interworking Instructions

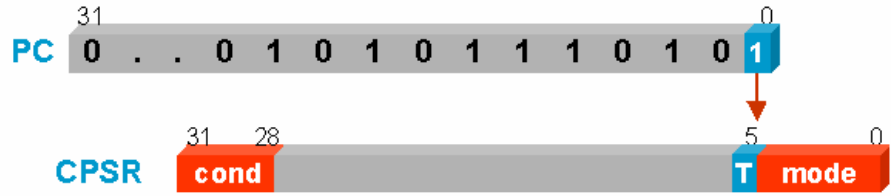
- Interworking is achieved
  - Using the Branch Exchange instructions
    - in Thumb state

      BX Rn
    - in ARM state (on Thumb-aware cores only)

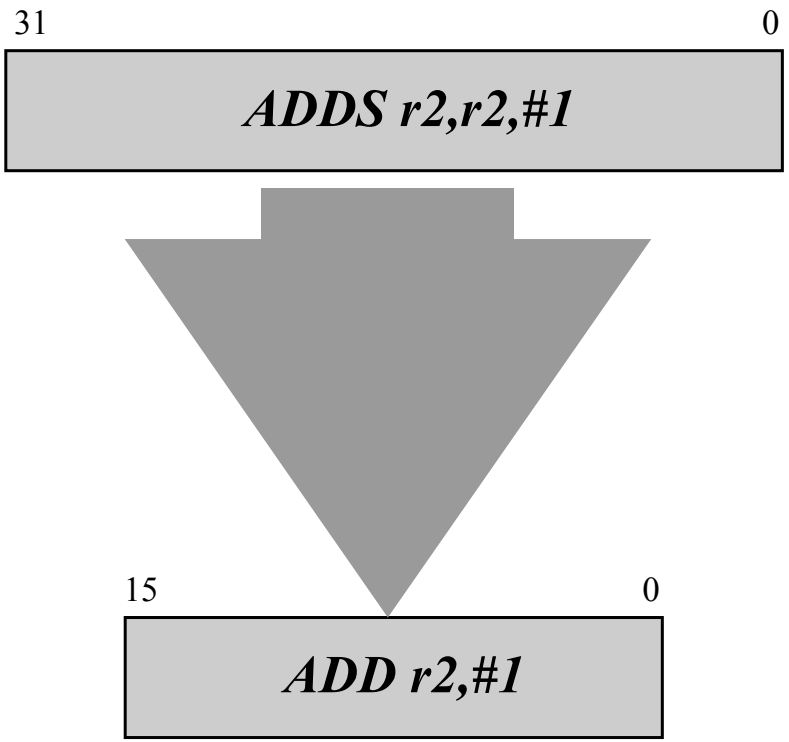
      BX<condition> Rn

    where Rn can be any registers (r0 to r15)
  - Using directive to specify the instruction type
    - Thumb : CODE16
    - ARM : CODE32

    CODE16 and CODE32 do not assemble to instructions that changes the state. They only instruct the assembler to assemble Thumb or ARM instructions.
- This BX instruction performs a branch to an absolute address in 4GB address space by copying Rn to the program counter
  - PC Bit0 =0 is ARM state
  - PC Bit0 =1 is Thumb state



ARM and Thumb instructions



32-bit ARM instruction

For most instruction generated by compiler:

- Conditional execution is not used
- Source and destination registers are identical
- Only Low registers are used
- Constants are of limited size
- Inline barrel shifter is not used

16-bit Thumb instruction

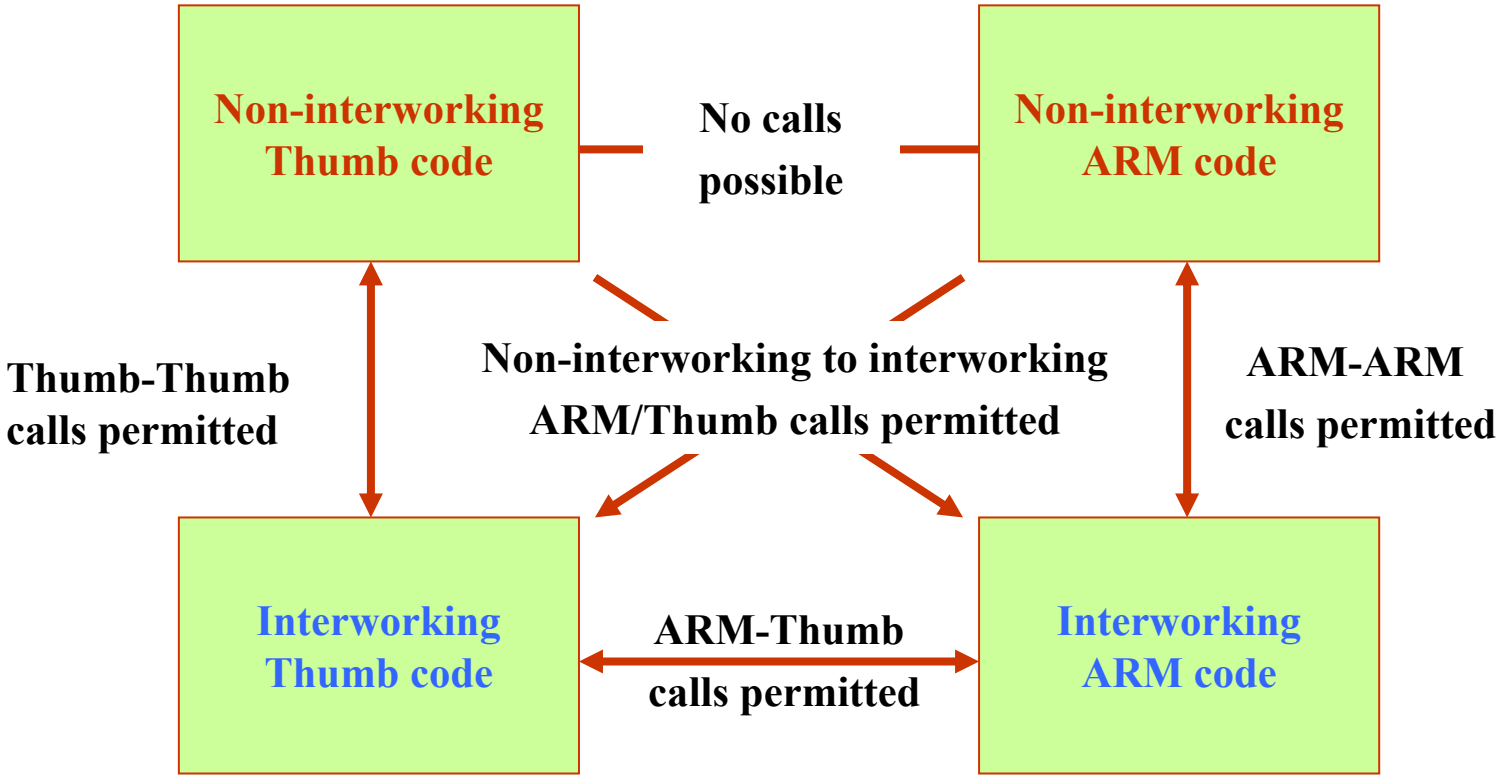
Example

```
;start off in ARM state
                    CODE32
                    ADR r0,Into_Thumb+1        ;generate branch target
                                               ;address & set bit 0,
                                               ;hence arrive Thumb state
                    BX r0                      ;branch exchange to Thumb
                    …
                    CODE16            ;assemble subsequent as Thumb
Into_Thumb     …
                    ADR r5,Back_to_ARM         ;generate branch target to
                                               ;word-aligned address,
                                               ;hence bit 0 is cleared.
                    BX r5                      ;branch exchange to ARM
                    …
                    CODE32            ;assemble subsequent as ARM
Back_to_ARM    …
```

ARM/Thumb Interworking between C/C++ and ASM

- C/C++ code compiled to run in one state may call assembler to execute in the other state, and vice-versa.
  - If the callee is in **C/C++**
    - compile it using **–apcs /interwork**
  - If the callee is in **ASM**
    - assemble it using **–apcs /interwork** and return using **BX LR**
- Assembler code used in this manner must conform to ATPCS where appropriate, e.g. function parameters passed in r0-r3 & r12 corruptible

Interworking Calls



Modules that are compiled for interworking generate slightly larger code, typically 2% larger for Thumb and less than 1% larger for ARM.

Outline

- Basic Code Development
- ARM/Thumb Interworking
- ***ARM/Thumb Procedure Call Standard***
- Lab1 – Code Development

ARM/Thumb Procedure Call standard

- Abbreviation
  - ATPCS
- Defines how subroutines can be separately written, separately compiled, and separately assembled to work together
- Describes a contract between a calling routine and a called routine
  - Obligations on the caller to create a memory state in which the called routine may start to execute.
  - Obligations on the called routine to preserve the memory-state of the caller across the call.
  - The rights of the called routine to alter the memory-state of its caller.

Goals of ATPCS

- Support Thumb-state and ARM-state equally.
- Support inter-working between Thumb-state and ARM-state.
- Favor:
  - Small code-size.
  - Functionality appropriate to embedded applications.
  - High performance.

    And where these aims conflict significantly, to standardize variants covering different priorities among them.
- Clearly distinguish between mandatory requirements and implementation discretion.
- Be binary compatible with:
  - The most commonly used variant of the previous APCS.
  - The most commonly used variant of the previous TPCS.

Machine Registers

- There are 16 registers of 32-bit wide visible to the ARM instruction set. These registers are labeled r0-r15 or R0-R15.
- The Thumb instruction set has full access rights to right general purpose registers r0-r7, and makes extensive use of r13-r15 for special purposes .
- r0-r3 are used to pass parameter values into a routine and result values out of a routine, and to hold intermediate values within a routine .
- In ARM-state, r12(IP) can also be used to hold intermediate values between subroutine calls.
- r4 to r11 are used to hold the values of a routine's local variables. They are also labeled v1-v8. Only v1-v4 can be used uniformly by the whole Thumb instruction set.
- r12-r15 have special roles. In these roles they are labeled IP, SP, LR and PC.
  - In ARM-state, register r12 (the alias of IP) can also be used to hold intermediate values *between* subroutine calls

Machine Register Table

Register	Synonym	Special	Role in the procedure call standard
r15		**PC**	The Program Counter.
r14		**LR**	The Link Register.
r13		**SP**	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8	FP	ARM-state variable-register 8. ARM-state frame pointer.
r10	v7	**SL**	ARM-state variable-register 7. Stack Limit pointer in stack-checked variants.
r9	v6	**SB**	ARM-state v-register 6. Static Base in PID,/re-entrant/shared-library variants
r8	v5		ARM-state variable-register 5.
r7	**v4**	**WR**	Variable register (v-register) 4. Thumb-state Work Register.
r6	**v3**		Variable register (v-register) 3.
r5	**v2**		Variable register (v-register) 2.
r4	**v1**		Variable register (v-register) 1.
r3	**a4**		Argument/result/scratch register 4.
r2	**a3**		Argument/result/ scratch register 3.
r1	**a2**		Argument/result/ scratch register 2.
r0	**a1**		Argument/result/ scratch register 1.

Veneer

- Veneer
  - A small block of code used with subroutine calls when there is a requirement to change processor state or branch to an address that cannot be reached in the current processor state.
- Interworking with veneer
  - The linker can link ARM code and Thumb code, and automatically generates interworking veneers to switch processor state when required. The linker also automatically generates long branch veneers, where required, to extend the range of branch instructions.
  - Armlink creates on input section call Veneer$$Code for each veneer.
  - You can see the information of veneer using
    - Armlink –o link_filename object_file_to_be_linked –info veneers

Outline

- Basic Code Development
- ARM/Thumb Interworking
- ARM/Thumb Procedure Call Standard
- ***Lab1 – Code Development***

Lab 1: Code Development

- Goal
  - How to create an application using ARM Developer Suite (ADS)
  - How to change between ARM state and Thumb state when writing code for different instruction sets
- Principles
  - Processor's organization
  - ARM/Thumb Procedure Call Standard (ATPCS)
- Guidance
  - Flow diagram of this Lab
  - Preconfigured project stationery files
- Steps
  - Basic software development (tool chain) flow
  - ARM/Thumb Interworking
- Requirements and Exercises
  - See note file
- Discussion
  - The advantages and disadvantages of ARM and Thumb instruction sets.

Lab 1: Code Development (cont')

- ARM/Thumb Interworking
  - Exercise 1: C/C++ for "Hello" program
    - Caller: Thumb
    - Callee: ARM
  - Exercise 2: Assembly for "SWAP" program, w/wo veneers
    - Caller: Thumb
    - Callee: ARM
  - Exercise 3: Mixed language for "SWAP" program, ATPCS for parameters passing
    - Caller: Thumb in Assembly
    - Callee: ARM in C/C++