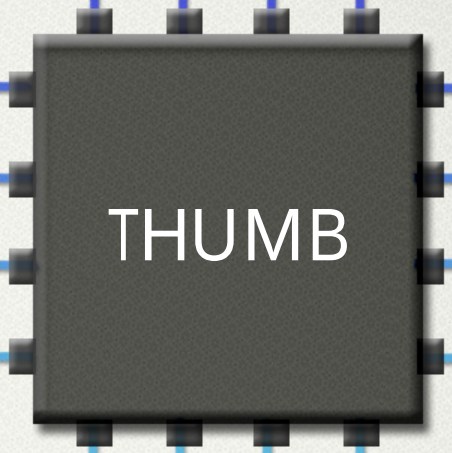


EE3002/IM2002 Microprocessor Part 2

THUMB

THUMB Instruction Set

- Key to reducing cost and increasing functionality
  - Put more code into fixed amount of memory
  - Put existing application into less memory
- Component cost lower if possible to execute same instruction on 16-bit memory
- However, ARM uses 32 bit instruction, so need 2 fetches to a 16-bit memory → reduce performance
- So it would incur less cost if 16-bit instruction is used
- Conversely, if instruction is 16-bit and memory is 32-bit wide will also uses less memory and save cost by reducing number of components together with less power
- This 16-bit instruction set is called THUMB

Solutions To Code-size Problem

- Hand code in assembler
- Improve the compiler
- Use compressed code (require additional system sources)
- ARM THUMB is a recoded subset of ARM instructions
- Implements 16-bit instruction on 32-bit architecture
- Keep 32-bit performance and address space
- A 30% code density improvement

Why THUMB?

- The biggest reason to look for an ARM processor with the THUMB instruction set is if you need to reduce code density
- In addition to reducing the total amount of memory required, you may also be able to narrow the data bus to just 16 bits
- With the narrower bus, it will take two bus cycles to fetch a single 32-bit instruction; but you'll only pay that penalty in the parts of your code that can't be implemented with the THUMB instructions
- And you'll still have the benefits of a powerful 32-bit RISC processor

THUMB introduction (1)

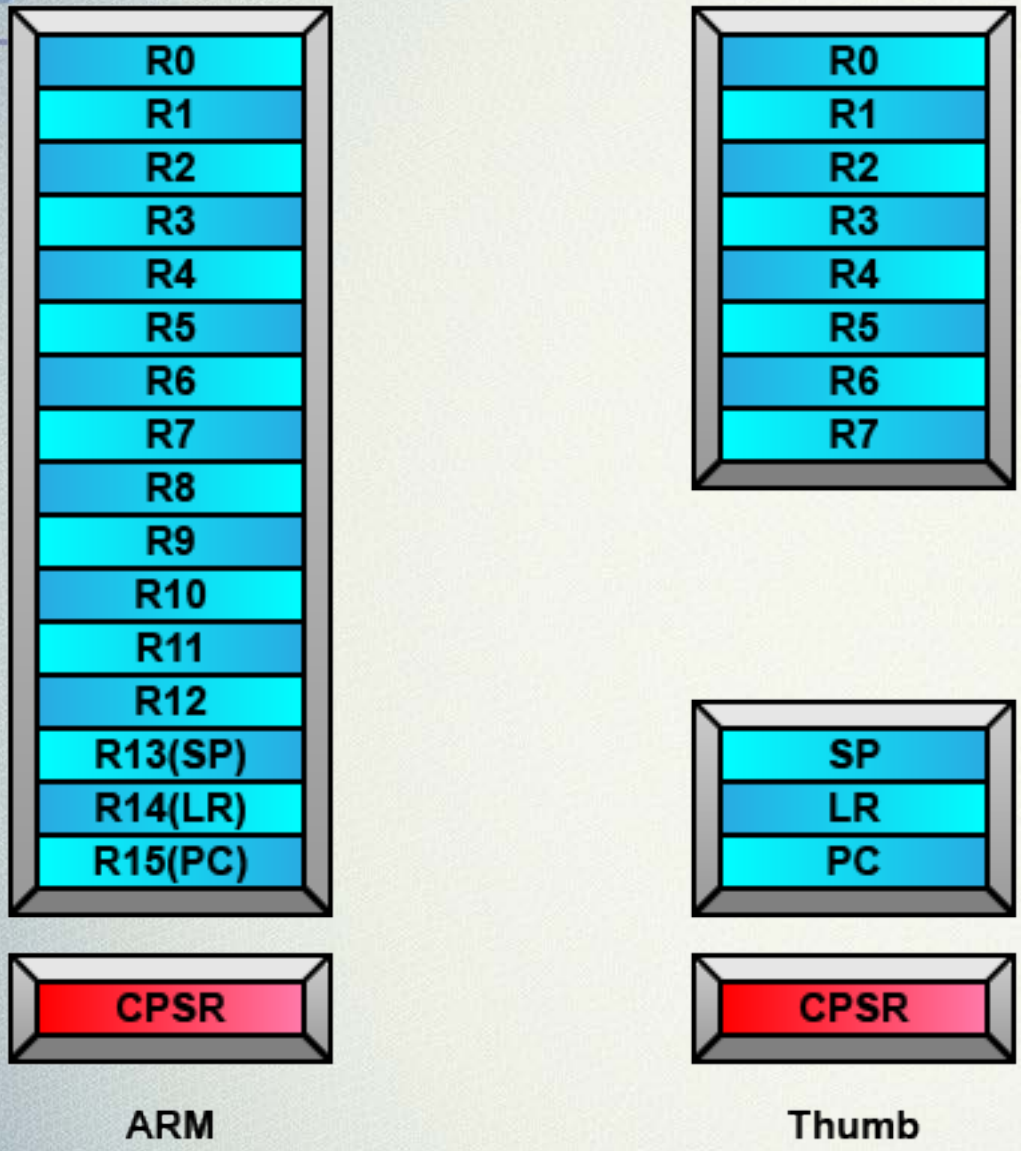
- The 16-bit THUMB instructions is a compact shorthand for a subset of the 32-bit instructions of the standard ARM
- Every THUMB instruction could be executed via the equivalent 32-bit ARM instruction
- However, not all ARM instructions are available in the THUMB subset; for example, there's no way to access status or coprocessor registers
- Also, some functions that can be accomplished in a single ARM instruction can only be simulated with a sequence of THUMB instructions

THUMB Introduction (2)

- ARM contains only one instruction set: the 32-bit set
- In *THUMB state*, the processor simply expands the smaller shorthand instructions fetched from memory into their 32-bit equivalents
- Difference between two equivalent instructions lies in how the instructions are fetched and interpreted prior to execution, not in how they function
- Since the expansion from 16-bit to 32-bit instruction is accomplished via dedicated hardware within the chip, it doesn't slow execution even a bit
- But the narrower 16-bit instructions do offer memory advantages

THUMB Introduction (3)

- THUMB instruction set do provide most of the functionality required in a typical application
- Arithmetic and logical operations, load/store data movements, and conditional and unconditional branches are supported
- Any code written in C could be executed successfully in THUMB state
- However, device drivers and exception handlers must often be written at least partly in ARM state.



Register Sets

When operating in the 16-bit THUMB state, the application encounters a slightly different set of registers

THUMB Register Usage

- In ARM, 17 registers are visible in user mode with one additional register, **SPSR** for exception mode only
- In THUMB
  - r0-r7: fully accessible
  - r8-r12: only accessible with MOV, ADD, CMP
  - r13, r14, r15: limited accessibility
  - cpsr/spsr: no direct access
  - (must switch to ARM to access)

Entering THUMB State (1)

- The usual method is via the Branch and Exchange (**BX**) instruction
  - BX *rd* ;(*rd contains target address*)
- Since all instructions will align themselves on either a 32- or 16-bit boundary, the LSB of the target address is always "0"
- Hence LSB of *rd* is ignored in determining the target address (assume to be ="0")
- Therefore this LSB in *rd is* free to be used for another purpose: to switch to THUMB state
  - if LSB = 1 when branching from ARM state, the processor switches to THUMB state before it begins executing from the new address
  - if LSB = 0 when branching from THUMB state, will switch back to ARM state

Entering THUMB State (2)

Bit 31 … LSB

0	1	0	0	0 … 0	1	0	0	1

Instruction address	Instruction
0x4000000C	…
0x40000008	MOV r0, #1
0x40000004	…
0x40000000	…

Branch to 0x40000008, ignoring LSB, assume =“0”

Switch to THUMB state since LSB == “1”

Entering THUMB State (3)

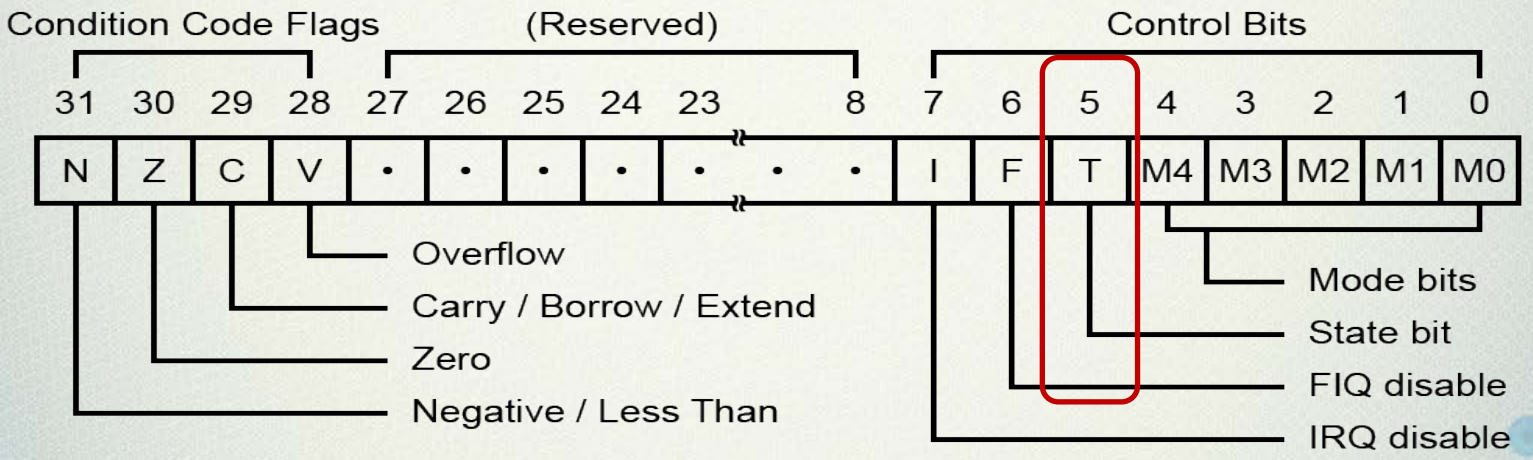
- When an exception occurs, the processor automatically begins executing in ARM state at the address of the exception vector
- So another way to change state is to place your 32-bit code in an exception handler
- If the CPU is running in THUMB state when that exception occurs, you can count on it being in ARM state within the handler
- If desired, you can have the exception handler put the CPU into THUMB state via a branch

CODE32 And CODE16 Directives

- These directives instruct the assembler to assemble subsequent instructions as ARM (CODE32) or THUMB (CODE16) instructions
- They do not switch the processor state at runtime
- You may have to insert instructions to switch state

Recollection of CPSR

- The **CPSR** register holds the processor mode (user or exception flag), interrupt mask bits, condition codes, and THUMB status bit
- The THUMB status bit (**T**) indicates the processor's current state: 0 for ARM state (default) or 1 for THUMB
- Although bits in the CPSR may be modified in software, it's dangerous to write to **T** directly which can produce unpredictable results



THUMB Instructions

- Size of ARM instruction can be reduced by examining the operands and bit fields
- Consider an ADD instruction, ADD r2, r2, #1 could be compressed easily
  - Destination = source → so can encode in same field
  - Encode #1 using 8 bits
  - Other restrictions, such as 3 bits for register r0-r7
  - Different instructions for higher registers r8 …

Further Restriction To ARM

- In addition to restricting the operands, other bits need to be taken into account as well such as the S bit and the conditional bits
- Conditional execution in instruction is removed
- However, branches can still be executed conditionally
- No inline barrel shifter option
- Individual instruction for shift and rotate is included

31 28	27 26	25	24 21	20	19 16	15 12	11 0
cond	0 0	1	opcode	S	Rn	Rd	Shifter_operand

Difference Between ARM And THUMB Instructions

1. **Data Processing instructions**

- THUMB instructions a subset of ARM instructions
- Separate shift instructions (eg., LSL, ASR, LSR, ROR)
- Many instructions only takes 2 operands (eg., ADD Rd, Rs)
- All instructions are unconditionally executed
- Condition code flags always updated when low registers are used (r0-r7)
- Most instructions only act on low registers with some exceptions such as CMP, MOV, and some variants of ADD and SUB
- A smaller range of constants

2. **SWI**

- The SWI number is limited to 8 instead of 24

Difference Between ARM And THUMB Instructions

3. There are no MSR or MRS instructions

- for cpsr/spsr

4. There are no coprocessor instructions

5. Branching ranges are shortened

- B<cond> label has a range of ±256 bytes
- B label has a range of ±2 KB
- BL label has a range of ±4 MB
- BX <Rd> is an absolute branch with optional state change

Difference Between ARM And THUMB Instructions

6. Single register load/store instructions are more restrictive

- LDR|STR <Rd>, [<Rn>, <offset>] is the only addressing mode allowed
- Two pre-indexed addressing modes are available
  - Base register + offset register
  - Base register + 5-bit offset
- Word, halfword and byte variants exist

Difference Between ARM And THUMB Instructions

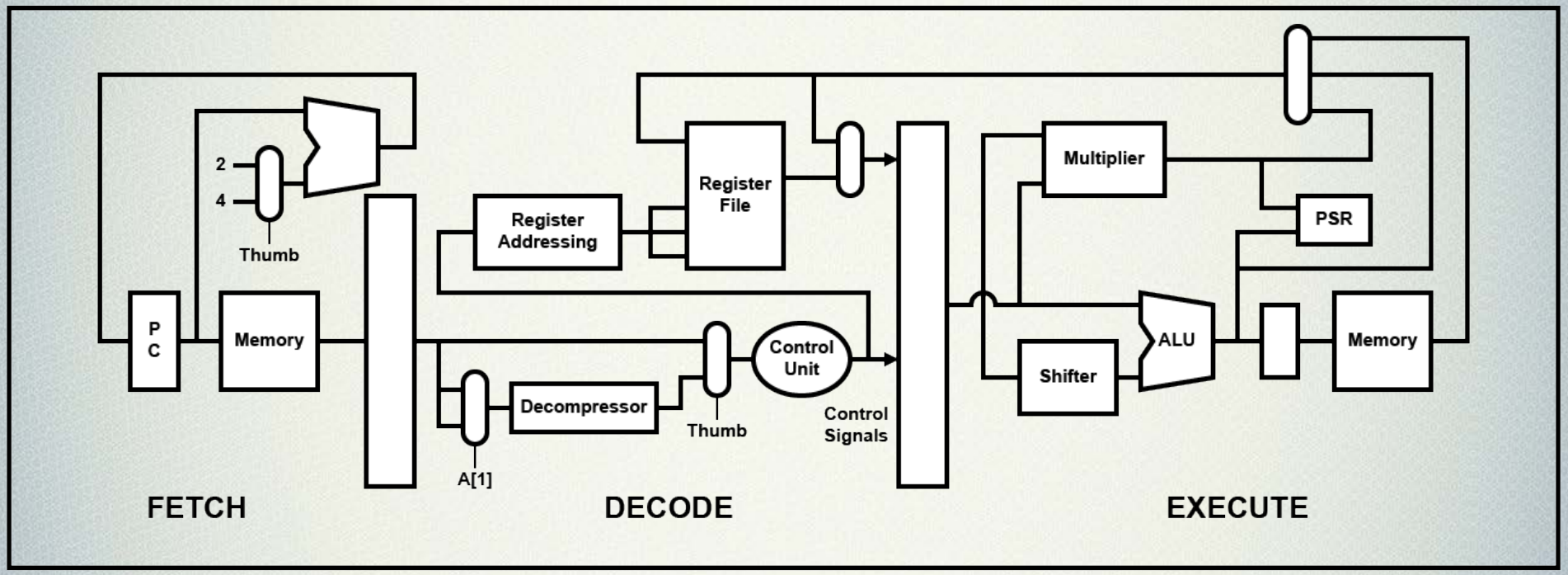
7. Load/store multiple instructions differ with 2 additional stack instructions

- LDMIA|STMIA <Rb>, <low reg list> can be used with low registers
- The stack operated by the PUSH and POP instructions is a Full Descending stack (FD)
  - PUSH {low-reg-list, lr} can be used to store registers and the link register on the stack
  - POP {low-reg-list, pc} can be used to load registers and the program counter from the stack

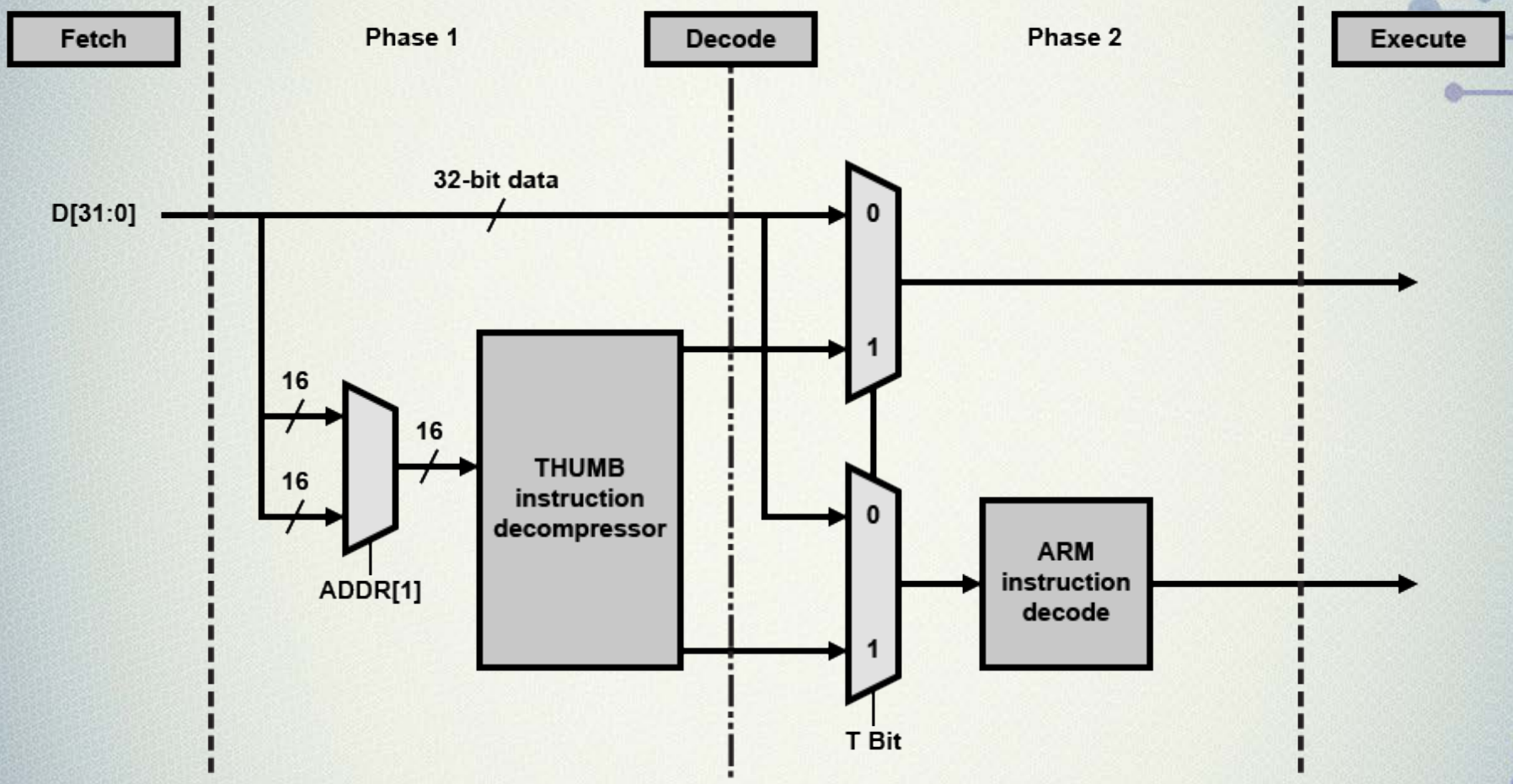
Implementation and Use

- Not a major concern for the programmers as the compiler do most of the work
- However good to understand some of the architectural modification
- The only part affected is the decode stage
- THUMB instruction expands to ARM instruction (decompression) via PLA
- No penalty incurred in decompression

Review - ARM Core and Pipeline



Processor Hardware



Example 1: Convert ARM To THUMB

- Give the mnemonic for a THUMB instruction that is equivalent to the ARM instruction
- **SUB r3, r2, r1, LSL #2**

<u>Answer</u>

- LSL r1, #2
- SUB r2, r1
- MOV r3, r2

<u>A better answer</u>

- MOV r0, r1 ;copy r1 to r0
- LSL r0, #2
- SUB r3, r2, r0

;Note: no change to r1, r2

Demo 1a: ARM Code

Project

- Target 1
  - Source Group 1

13armCode.s

```
        AREA armCode, CODE, READONLY
        ENTRY
        CODE32                  ;following is arm code
        MOV r1, #2
        MOV r2, #10
        SUB r3, r2, r1, LSL #2
stop    B   stop
        END
```

Build Output

```
Build target 'Target 1'
assembling 13armCode.s...
linking...
Program Size: Code=16 RO-data=0 RW-data=0 ZI-data=0
"13armCode.axf"    0 Error(s), 0 Warning(s).
```

Note: code size = 16 bytes (ARM codes, 4*4 = 16)

Demo 1b: THUMB Code

Project

Target 1

Source G

13thumbCode.s

```
01          AREA TothumbCode, CODE, READONLY
02          ENTRY
03          CODE32                      ;following is arm code
04          LDR r0, =thumbCode + 1      ;+1 to set LSB to 1
05          BX  r0                      ;call thumb subroutine
06
07          CODE16
08  thumbCode                           ;following is thumb code
09          MOV r1, #2
10          MOV r2, #10
11          LSL r1, #2
12          SUB r2, r1
13          MOV r3, r2
14  stop    B   stop
15          END
```

Arm code size = 2*4 = 8 bytes

THUMB code size = 6*2 = 12 bytes

Build Output

```
Build target 'Target 1'
assembling 13thumbCode.s...
linking...
Program Size: Code=24 RO-data=0 RW-data=0 ZI-data=0
"13thumbCode.axf" - 0 Error(s), 0 Warning(s).
```

Total code size = 8(arm) + 12(thumb) + 4(literal) = 24 bytes

Demo 1b: THUMB Code (Check T Bit For Arm State)

Register	Value
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000004
CPSR	0x000000D3
N	0
Z	0
C	0
V	0
Q	0
I	1
F	1
T	0
M	0x13
SPSR	0x00000000
User/System	

Project | Registers

```
6:
7:                    CODE16
8: thumbCode                                          ;
0x00000004  E12FFF10  BX        R0
```

13thumbCode.s

```
01          AREA TothumbCode, CODE, READONLY
02          ENTRY
03          CODE32                      ;following is
04          LDR r0, =thumbCode + 1      ;+1 to set LS
05          BX  r0                      ;call thumb s
06
07          CODE16
08 thumbCode                            ;following is
09          MOV r1, #2
10          MOV r2, #10
11          LSL r1, #2
12          SUB r2, r1
13          MOV r3, r2
14 stop     B   stop
15          END
```

T bit = 0 → arm state

Demo 1b: THUMB Code (Check T Bit For Thumb State)

Register	Value
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000008
CPSR	0x000000F3
N	0
Z	0
C	0
V	0
Q	0
I	1
F	1
T	1
M	0x13
SPSR	0x00000000
User/System	

Project | Registers

```
     9:                    MOV r1, #2
0x00000008  2102       MOV        R1,#0x02
    10:                    MOV r2, #10
0x0000000A  220A       MOV        R2,#0x0A
```

13thumbCode.s

```
01          AREA TothumbCode, CODE, READONLY
02          ENTRY
03          CODE32                          ;following is
04          LDR r0, =thumbCode + 1          ;+1 to set LS
05          BX  r0                          ;call thumb s
06
07          CODE16
08  thumbCode                               ;following is
09          MOV r1, #2
10          MOV r2, #10
11          LSL r1, #2
12          SUB r2, r1
13          MOV r3, r2
14  stop    B   stop
15          END
```

T bit = 1 → thumb state

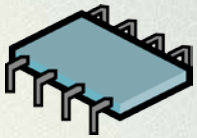
Switching Between THUMB And ARM

- Bulk of C\C++ instructions are compiled to THUMB instructions for performance and code density
- However, at times it might be necessary to switch to ARM instructions
- Examples
  - Speed-critical parts of an application, compression algorithms
  - Certain operations cannot be done in THUMB, enable/disable interrupt
  - Some standalone programs
  - Hence a need to switch between the two states

Switching Between THUMB And ARM

- Use *BX Rn* instruction
- To compute target address
  - Register Rn contains the target address
  - Target address ignores what is the LSB (bit 0th) value, always assume =“0”
- To switch state, look at the LSB of register Rn
  - If LSB is “0”, the state is switch to ARM
  - If LSB is “1”, the state is switch to THUMB

Example 2: Branch from ARM to THUMB and back to ARM



```
AREA arm2thumb, CODE, READONLY
ENTRY
• main
LDR r0, =thumbProg + 1      ;+1 to set thumb mode
BX   r0                     ;branch to thumbProg
CODE16                      ;following are THUMB code
• thumbProg
MOV r2, #20
MOV r3, #30
ADD  r2, r3
LDR  r0, =armProg           ;LSB of addr of instr = 0, ARM state
BX    r0                    ;set to ARM state
```

Example 2: Continue

```
CODE32      ;following are ARM code
armProg
MOV r4, #40
MOV r5, #50
ADD  r5, r5, r4
stop
B      stop
END
```

Demo: ARM → THUMB → ARM

Project

- Target 1
  - Source Group 1

13arm2thumb.s

```
01      AREA arm2thumb, CODE, READONLY
02      ENTRY
03 main
04      LDR r0, =thumbProg + 1   ;+1 to set thumb mode
05      BX  r0                   ;branch to thumbProg
06      CODE16
07 thumbProg
08      MOV r2, #20
09      MOV r3, #30
10      ADD r2, r3
11      LDR r0, =armProg         ;LSB of addr of instr = 0
12      BX  r0                   ;set T bit to 0 arm state
13      CODE32
14 armProg
15      MOV r4, #40
16      MOV r5, #50
17      SUB r5, r5, r4
18 stop
19      B   stop
20      END
```

Build Output

```
assembling 13arm2thumb.s...
13arm2thumb.s(13): warning: A1581W: Added 2 bytes of padding at address 0x12
13arm2thumb.s:    13 00000014  CODE32
13arm2thumb.s: 0 Errors, 1 Warning
linking...
Program Size: Code=44 RO-data=0 RW-data=0 ZI-data=0
"13arm2thumb.axf" - 0 Error(s), 1 Warning(s).
```

Demo 2: ARM To THUMB (Starts In Arm State)

Registers

Register	Value
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000004
CPSR	0x000000D3
N	0
Z	0
C	0
V	0
I	1
F	1
T	0
M	0x13

arm state

Disassembly

```
5:          BX   r0          ;branch to thumbProg
6:          CODE16
7: thumbProg
0x00000004  E12FFF10  BX        R0
```

13arm2thumb.s

```
01      AREA arm2thumb, CODE, READONLY
02      ENTRY
03 main
04      LDR r0, =thumbProg + 1   ;+1 to set tl
05      BX  r0                   ;branch to tl
06      CODE16
07 thumbProg
08      MOV r2, #20
09      MOV r3, #30
10      ADD r2, r3
11      LDR r0, =armProg         ;LSB of addr
        BX  r0                   ;set T bit to
        CODE32
        Prog
```

Demo 2: ARM To THUMB (Thumb State)

Registers

Register	Value
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000008
CPSR	0x000000F3
N	0
Z	0
C	0
V	0
I	1
F	1
T	1
M	0x13

thumb state

Disassembly

```
     8:          MOV r2, #20
0x00000008  2214       MOV        R2,#0x14
     9:          MOV r3, #30
0x0000000A  231E       MOV        R3,#0x1E
```

13arm2thumb.s

```
01      AREA arm2thumb, CODE, READONLY
02      ENTRY
03  main
04      LDR r0, =thumbProg + 1   ;+1 to set tl
05      BX  r0                   ;branch to tl
06      CODE16
07  thumbProg
08      MOV r2, #20
09      MOV r3, #30
10      ADD r2, r3
11      LDR r0, =armProg         ;LSB of addr
12      BX  r0                   ;set T bit to
13      CODE32
14  armProg
```

Demo 2: ARM to THUMB (arm state again)

Register	Value
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000014
CPSR	0x000000D3
N	0
Z	0
C	0
V	0
I	1
F	1
T	0
M	0x13

arm state again

Project | Registers

```
    15:          MOV r4, #40
0x00000014  E3A04028  MOV        R4,#0x00000028
    16:          MOV r5, #50
0x00000018  E3A05032  MOV        R5,#0x00000032
```

13arm2thumb.s

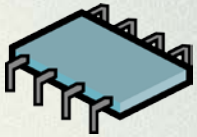
```
02      ENTRY
03 main
04      LDR r0, =thumbProg + 1   ;+1 to set t
05      BX  r0                   ;branch to t
06      CODE16
07 thumbProg
08      MOV r2, #20
09      MOV r3, #30
10      ADD r2, r3
11      LDR r0, =armProg         ;LSB of addr
12      BX  r0                   ;set T bit t
13      CODE32
14 armProg
15      MOV r4, #40
```

Jump To THUMB Subroutine And Return From It

- **Need to insert codes called veneer**

1. Call veneer and switch to THUMB state
2. Jump to THUMB subroutine
3. Upon return from THUMB subroutine, switch back to ARM state

Example 3: Calling THUMB Subroutine



```
        AREA ToThumbSub, CODE, READONLY
        ENTRY
main
        BL  veneer    ;call veneer
stopB   stop   ;return from thumb sub
veneer   ;insert codes to switch to thumb and jump to thumb sub
        LDR r0, =thumbSub+1
        BX    r0    ;jump to thumb sub and switch to thumb
        CODE16
thumbSub
        MOV r4, #40
        MOV r5, #50
        SUB r5, r5, r4
        BX  lr      ;return to main and switch back to ARM
        END
```

Demo 3: Calling THUMB Subroutine (1)

Registers

Register	Value
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000000
R15 (PC)	0x00000000
CPSR	0x000000D3
N	0
Z	0
C	0
V	0
I	1
F	1
T	0
M	0x13

13thumbSub.s

```
01          AREA ToThumbSub, CODE, READONLY
02          ENTRY
03  main
04          BL   veneer   ;call veneer
05  stop    B    stop     ;return from thumb sub
06  veneer  ;insert codes to switch to thumb and jump to thumb sub
07          LDR r0, =thumbSub+1
08          BX  r0        ;jump to thumb sub and switch to thumb
09          CODE16
10  thumbSub
11          MOV r4, #40
12          MOV r5, #50
13          SUB r5, r5, r4
14          BX  lr        ;return to main and switch back to ARM
15          END
```

Starts in ARM state

Demo 3: Call THUMB Subroutine (2)

Registers

Register	Value
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000004
R15 (PC)	0x00000010
CPSR	0x000000F3
N	0
Z	0
C	0
V	0
I	1
F	1
T	1
M	0x13

13thumbSub.s

```
01          AREA ToThumbSub, CODE, READONLY
02          ENTRY
03  main
04          BL  veneer   ;call veneer
05  stop    B   stop     ;return from thumb sub
06  veneer  ;insert codes to switch to thumb and jump to thumb sub
07          LDR r0, =thumbSub+1
08          BX  r0       ;jump to thumb sub and switch to thumb
09          CODE16
10  thumbSub
11          MOV r4, #40
12          MOV r5, #50
13          SUB r5, r5, r4
14          BX  lr       ;return to main and switch back to ARM
15          END
```

Switch to THUMB state

Demo 3: Call THUMB Subroutine (3)

Registers

Register	Value
R10	0x00000000
R11	0x00000000
R12	0x00000000
R13 (SP)	0x00000000
R14 (LR)	0x00000004
R15 (PC)	0x00000004
CPSR	0x200000D3
N	0
Z	0
C	1
V	0
I	1
F	1
T	0
M	0x13

Project | Registers

13thumbSub.s

```
01          AREA ToThumbSub, CODE, READONLY
02          ENTRY
03  main
04          BL   veneer   ;call veneer
05  stop    B    stop     ;return from thumb sub
06  veneer  ;insert codes to switch to thumb and jump to thumb sub
07          LDR r0, =thumbSub+1
08          BX  r0        ;jump to thumb sub and switch to thumb
09          CODE16
10  thumbSub
11          MOV r4, #40
12          MOV r5, #50
13          SUB r5, r5, r4
14          BX  lr        ;return to main and switch back to ARM
15          END
```

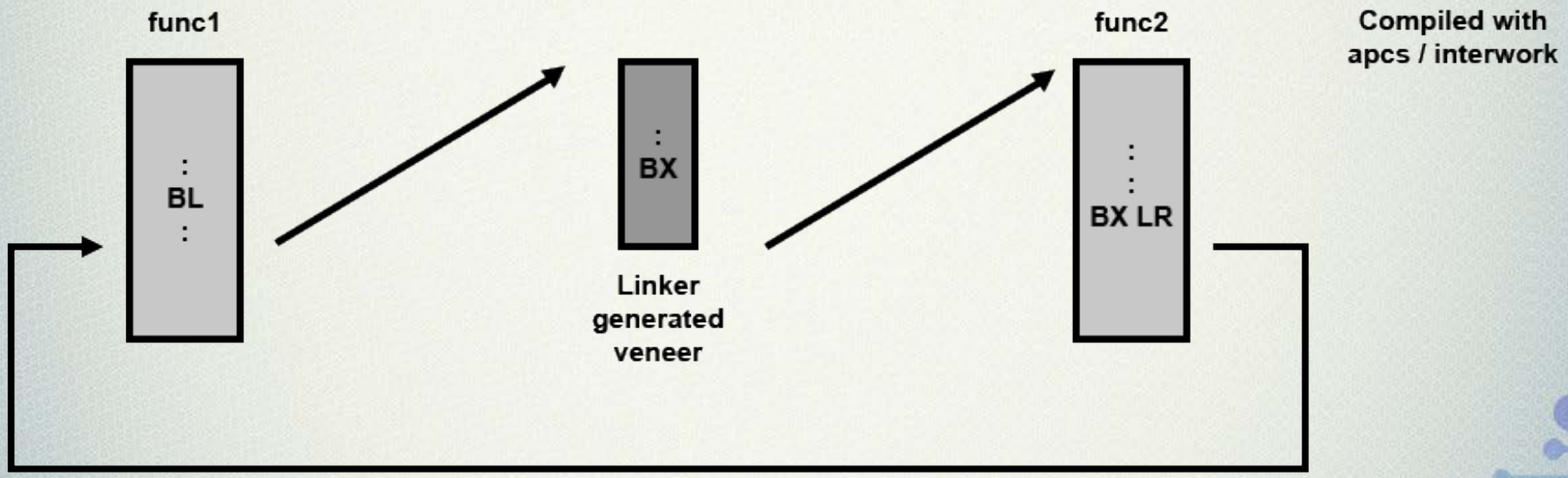
Return from THUMB subroutine and switch back to ARM state

Veener by linker

- For a mixture of source files written separately in ARM and THUMB codes, linker veneers are inserted
- Linker veneers are small sections of code generated by the linker and inserted into your program
- The purpose of a linker veneer
  - Extend the range of a branch by becoming the intermediate target of the instruction and then setting the PC to the destination address
  - **If ARM and THUMB are mixed, the veneer also changes processor state**

Veener For ARM/THUMB Interworking

- func1 compiled for ARM
- func2 compiled for THUMB
- Veener will be generated by linker to switch state



Example 4:
ARM/THUMB Interworking Using asm
(With Linker Veneer)

arm.s	thumb.s

AREA arm, CODE, READONLY
IMPORT thumbProg
ENTRY
armProg
MOV r0, #1
BL thumbProg
MOV r2, #3
stop
B stop
END
	AREA thumb, CODE, READONLY
	CODE16
	EXPORT thumbProg
	thumbProg
	MOV r1, #2
	BX lr
	END

Note: 2 separate files, thumb main → thumb sub

Demo 4: ARM/THUMB interworking using asm (with linker veneer)

Registers

Register	Value
Current	
R0	0x00000001
R1	0x00000000
R2	0x00000000
R3	0x00000000
R4	0x00000000
R5	0x00000000
R6	0x00000000
R7	0x00000000
R8	0x00000000
R9	0x00000000
R10	0x00000000
R11	0x00000000
R12	0x00000019
R13 (SP)	0x00000000
R14 (LR)	0x00000008
R15 (PC)	0x00000014
CPSR	0x000000D3
SPSR	0x00000000
User/System	

Project | Registers

Disassembly

```
0x0000000C  EAFFFFFE  B       0x0000000C
0x00000010  E28FC001  ADD     R12,PC,#0x00000001
0x00000014  E12FFF1C  BX      R12
      5:          MOV r1, #2
0x00000018  47702102  LDRMIB  R2,[R0,-R2,LSL #2]!
0x0000001C  00000000  ANDEQ   R0,R0,R0
0x00000020  00000000  ANDEQ   R0,R0,R0
0x00000024  00000000  ANDEQ   R0,R0,R0
```

13arm.s | 13thumb.s

```
01      AREA arm, CODE, READONLY
02      IMPORT thumbProg
03      ENTRY
04 armProg
05      MOV r0, #1
06      BL      thumbProg
07      MOV r2, #3
08 stop
09      B stop
10      END
```

Linker generated veneer

Code Size: ARM Vs THUMB

C CODE

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

arm CODE

```
CMP    r0, #0          ;compare r0 to zero
RSBLT r0, r0, #0;if r0<0, then do r0=0-r0 (-x)
MOV   pc, lr           ;return
```

Size:
3*4 =12 bytes

thumb CODE

```
CODE16

CMP    r0, #0          ;compare r0 to zero
BGE    rtn        ;jump to rtn if >= 0
NEG    r0, r0          ;else negate (-x)
rtn
MOV   pc, lr           ;return
```

Size:
4*2 = 8 bytes

Summary

- Why THUMB?
- Entering thumb state
- BL, BX instructions and T bit
- Converting arm instructions to thumb instructions and vice versa
- Insert code (veneer) to switch between arm/thumb subroutine
- Linker generated veneer
- Code size