Armv8-M Mainline Assembly Programming

Agenda

Introduction

Data Processing Instructions

Load/Store Instructions

Flow Control

Miscellaneous

Arm Custom Instructions

Learning objectives - Introduction

After completing this section, you will be able to:

- Discuss the situations in which assembly language programming is required
- Describe the syntax of assembly language instructions in the T32 instruction set

Before we start...

This module provides an introduction to Armv8-M assembly language

- This module is NOT a complete summary of available instructions

For further information on the Armv8-M instruction set, see

- The Armv8-M Architecture Reference Manual
- The Arm Compiler toolchain – armasm User Guide

Arm Compiler 6 supports Armv8-M and provides two assemblers:

- **`armasm`** – legacy assembler used in Arm Compiler 5
- **`armclang`** – assembler and C/C++ compiler

Terminology – “Arm” can mean a number of different things, for example

- The company
- The processor core
- The general architecture

Why do you need to know assembler?

Armv8-M processors can be programmed using C

- It is not usually necessary to use assembly code
- Exception model automatically saves the thread context

However, assembler is sometimes needed

- Validation work, or testing corner-case behavior
- Reset handler, operating systems, device drivers or other hand-optimized critical code
- Very helpful to understand the instruction set while debugging

Some features of the Arm architecture may not be available through the compiler

- CMSIS provides intrinsics to access certain instructions
- Example: `__WFI();` (implements `WFI` instruction)

Some compiled code can be optimized by coding in assembly

Instruction set basics

The Arm Architecture is a Load/Store architecture

- Data must be loaded from memory into the CPU, modified, then written back out
- No direct manipulation of memory contents

Instructions consist of

- Opcode, destination register, first source operand, optional second source operand

```
OPCODE{<qualifier>}{<cond>} Rd, Rn, {Rm}
```

T32 Instruction Set

- Mix of 16-bit and 32-bit instructions
- Superset of the traditional 16-bit Thumb instruction set
- Optimized for code density from C code
- Baseline architecture's instruction set is derived from the Armv6-M Thumb instruction set
  - Limited use of registers, constants and conditional execution
- Main Extension is derived from the Armv7-M architecture and adds more instructions

Unified Assembler Language (UAL)

UAL gives the ability to write assembler code for all Arm processors

- Previously code had to be written exclusively for Arm state (not available in Armv8-M) or Thumb state
- UAL allows the execution state to be decided at assembly time
- Legacy assembler code will still assemble successfully

UAL also defines ‘pseudo’ instructions that are resolved by the Assembler

- The assembler will generate the machine code dependent upon the inline directives (e.g. `.thumb`) or the assembler switches (e.g. `-mcpu`)

Some general rules for UAL

- Use of `POP`, `PUSH`
- Relaxation of register definitions for `Rd` and `Rn`
- Requirement of `S` to enable flag setting

See complete definition in the Armv8-M Architecture Reference Manual

Condition codes and flags (1)

Condition codes can be used for conditional execution of assembly instructions

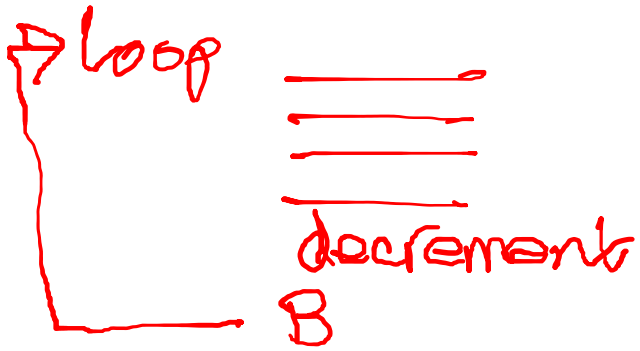
The codes evaluate as `TRUE` or `FALSE` based on the values of the condition flags

The condition flags are part of the Application Program Status Register (APSR)

Certain assembly instructions set the condition flags

- For some instructions that do not set the flags by default, the `S` qualifier can be added

APSR bit	Condition flag	Meaning
31	N	Negative
30	Z	Zero
29	C	Carry
28	V	Overflow



Condition codes and flags (2)

Condition code	Meaning	Flags
EQ	Equal	Z set
NE	Not equal	Z clear
CS or HS	Carry set, or Higher or same (unsigned >=)	C set
CC or LO	Carry clear, or lower (unsigned <)	C clear
MI	Negative	N set
PL	Positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Higher (unsigned **>**)	C set **AND** Z clear
LS	Lower or same (unsigned **<=**)	C clear **OR** Z set
GE	Signed **>=**	N and V the same
LT	Signed **<**	N and V differ
GT	Signed **>**	Z clear **AND** N and V the same
LE	Signed **<=**	Z set **AND** N and V differ
AL	Always. This is the default, and it normally omitted.	Any

Thumb instruction encoding choice

When assembling for an Armv8-M Mainline processor there is often a choice of 16-bit and 32-bit instruction encodings

- The assembler will normally generate 16-bit instructions

Instruction width specifiers

- Allow you to determine which instruction width the assembler will use
- Can be placed immediately after instruction mnemonics:
  - `.W`
    - Forces a 32-bit instruction encoding
  - `.N`
    - Forces a 16-bit instruction encoding
- Errors raised by assembler if not possible

Disassembly rules

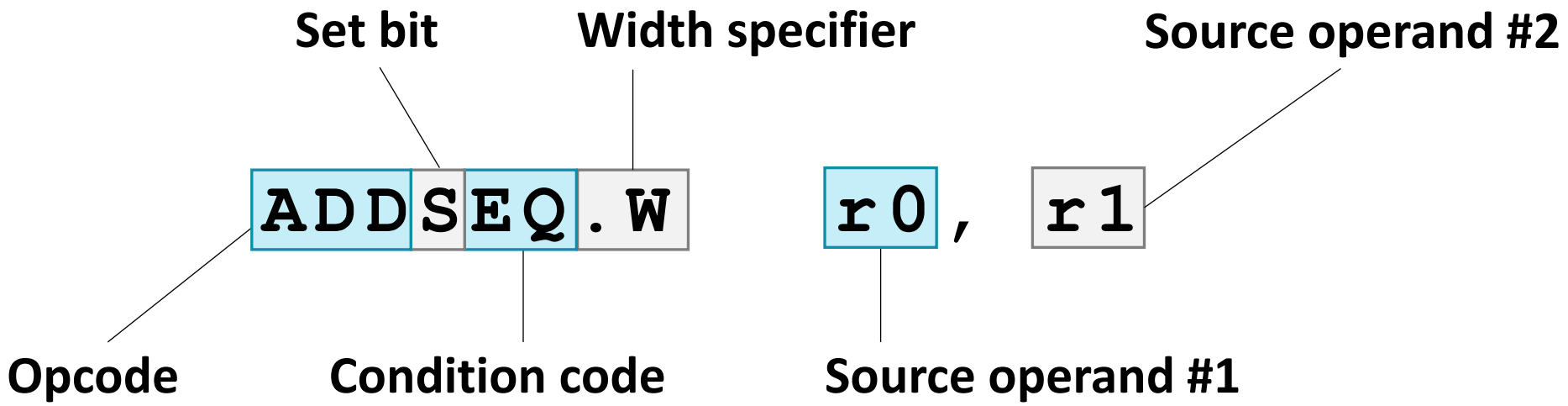
- One-to-one mapping is defined to ensure correct re-assembly
- `.W` or `.N` suffix used for cases when a bit pattern which doesn't follow the above rules is disassembled

Section quiz - Introduction

What are the different components of this instruction?

Section quiz - Introduction

What are the different components of this instruction?



Agenda

Introduction

Data Processing Instructions

Load/Store Instructions

Flow Control

Miscellaneous

Arm Custom Instructions

Learning objectives - Data processing instructions

After completing this section, you will be able to:

- Discuss why being able to understand assembly language instruction sequences is useful
- Compare common C language source code sequences to their corresponding T32 instruction sequences

Data processing instructions (1)

These instructions operate on the contents of registers

- They DO NOT affect memory
- Comparison instructions only set the condition code flags
- Data processing instructions set the condition code flags only if suffix 'S' is added, for example:

```
ADDS r0, r1, r2  // r0 = r1 + r2
```

- Conditional execution – uses the `IT` Instruction (discussed later)

	arithmetic		logical		move
manipulation (has destination register)	ADC ADD	SBC SUB RSB	BIC AND	EOR ORR ORN	MVN MOV
comparison (set flags only)	CMN (ADDS)	CMP (SUBS)	TST (ANDS)	TEQ (EORS)	

Data processing instructions (2)

Arithmetic instruction	Operation	Flags set?	Result saved?
`ADDS r0, r1, r2`	`r0 = r1 + r2`	Yes, all	Yes
`ADCS r0, r1`	`r0 = r0 + r1 + <carry_flag>`	Yes, all	Yes
`SUB r3, r1, r7`	`r3 = r1 - r7`	No	Yes
`RSB r3, r3, r7`	`r3 = r7 - r3`	No	Yes
`CMP r1, r7`	`r1 - r7`	Yes, all	No

Logical instruction	Operation	Flags set?	Result saved?
`ANDS r0, r1, #0xA0`	`r0 = r1 & 0xA0`	Only N, Z	Yes
`BIC r0, r1, #0xA0`	`r0 = r1` with bits `5` and `7` cleared	No	Yes
`ORRS r0, r1, #0xA0`	`r0 = r1 | #0xA0`	Only N, Z	Yes
`TST r1, #0xAB`	`r1 & 0xAB`	Only N, Z	No

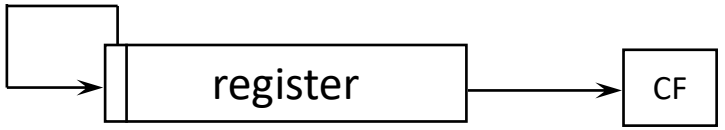
Generating data processing instructions

```
dest1 = op1 + op2;
dest2 = op3 - op4;

if(dest1 > dest2)
{
    dest1 = op5 & 0xAA;
}
else
{
    dest2 = op6 | 0x55;
}
```

```
    adds    r0, r1, r0   // r0 = r1 + r0
    subs    r2, r2, r3   // r2 = r2 - r3
    cmp     r0, r2       // same as SUB but
                         // only affects APSR
    ble     .LBB0_1
    movs    r4, #170     // r4 = 0xAA
    ands    r0, r4       // r0 = r0 & r4
    b       .LBB0_2
.LBB0_1:
    movs    r5, #85      // r5 = 0x55
    orrs    r2, r5       // r2 = r2 | r5
.LBB0_2:
```

Shift operations

ASR: Arithmetic Shift Right



Division by a power of 2,
preserving the sign bit

```
int dest1;
…
dest1 = dest1 >> 4;
        ASR r8,r8,#4
```

LSL: Logical Shift Left

Multiplication by a power of 2

```
int dest2;
…
dest2 = dest2 << 8;
        LSL r9,r9,#8
```

LSR: Logical Shift Right

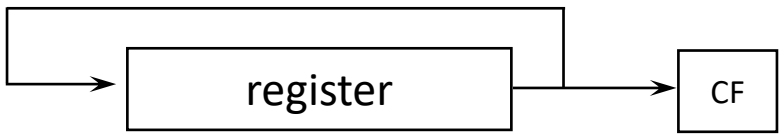
Division by a power of 2

```
unsigned int dest3;
…
dest3 = dest3 >> 4;
        LSR r10,r10,#4
```

These are also available as part of the flexible second operand

Rotate operations

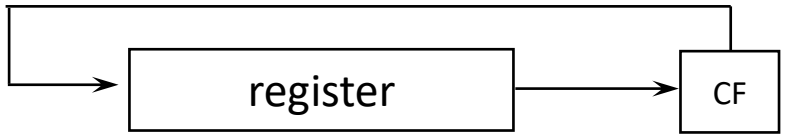
ROR: Rotate Right



Bit rotate with wrap around
from LSB to MSB

```
dest1 = __ROR(op1,4);
        ROR r8,r4,#4
```

RRX: Rotate Right Extended



Single bit rotate with wrap around
from CF to MSB

```
dest2 = __RRX(op1);
        RRX r0,r0
```

These are also available as part of the flexible second operand

Flexible second operand - Registers

For many instructions the second operand is flexible

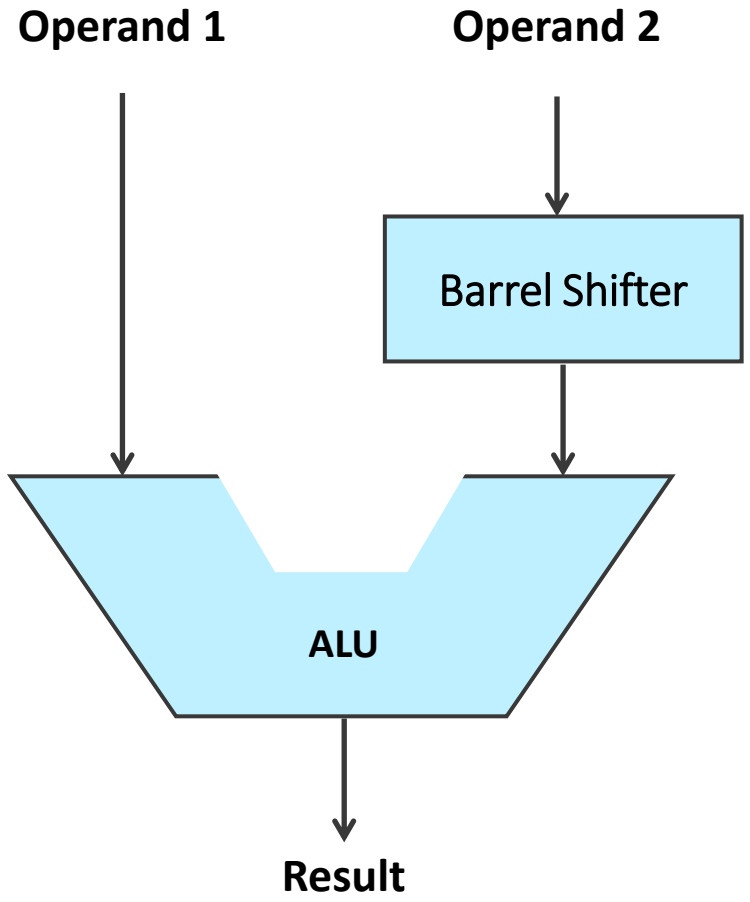
- Either a register with optional shift
- Or an immediate constant

Register, with optional shift

- Shift distance can be a 5-bit unsigned integer

Can be used for multiplication by constant, for example:

```
int op1, op2, dest1, dest2;
…
dest1 = op1 * 3;
        ADD r8, r4,      r4,LSL #1
                op1      2 * op1
dest2 = dest1 – op2 * 8;
        SUB r9, r8,      r5,LSL #3
                dest1    3 * op2
```



Flexible second operand - Constants (1)

There is limited space for constants because of fixed-length instructions

Constants can be

- 8-bit number shifted left by any number of places
- In the form `0x00XY00XY`, `0xXY00XY00` or `0xXYXYXYXY`

Some instructions have a special immediate format

- `MOVW` loads a 16-bit immediate value into the lower half of the register, and clears the other half, for example:
  - `MOVW Rd, #imm16`
- `MOVT` loads a 16-bit immediate value into the upper half of the register
- `ADDW` and `SUBW` can use any 12-bit positive constant, for example:
  - `ADDW Rd, Rn, #imm12`

Attempts to use constants which are not in the correct range will generate an assembly error

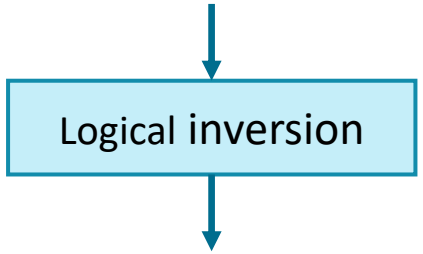
- Use the `LDR` pseudo-op – `LDR Rn, =<constant>`
- Assembler will use optimal sequence to generate constant into specified register

Flexible second operand - Constants (2)

Can still encode other constants through *instruction substitutions*

- The assembler performs substitutions automatically

```
ADCS r0, r0, #0xFFFFFFF0
     r0 = r0 + (-16)
```




```
SBCS r0, r0, #0xF
     r0 = r0 - 16
```

8-bit?	12-bit?	Pattern-match	ROR?
✗	✗	✗	✗

8-bit?	12-bit?	Pattern-match	ROR?
✓	✓	✗	✓

- **SBCS** uses the same architectural pseudocode function as **ADCS**

```
(result, carry, overflow) = AddWithCarry(R[n], NOT(imm32), APSR.C);
```

Loading constants into registers

Compilers use optimal sequence to generate constant into specified register

- **MOV**, **MVN**, **MOVW**/**MOVT**, or **LDR** from a literal pool
- Constant determined at compile or link time

Examples:

```
int op1, op2, op3, op4;

          ...
op1 = 0x2543;
          MOV  r0,#0x2543
op2 = 0xFFFF43FF;
          MVN  r1,#0x0000bc00
op3 = 0x2F008000;
          MOVW r7,#0x8000
          MOVT r7,#0x2F00
op4 = 0xFFFFF5;
          LDR  r3,.LCPI0_0
          ...
.LCPI0_0
          .long 16777205   @ 0xfffff5
```

Execute-only support: (`-mexecute-only`)

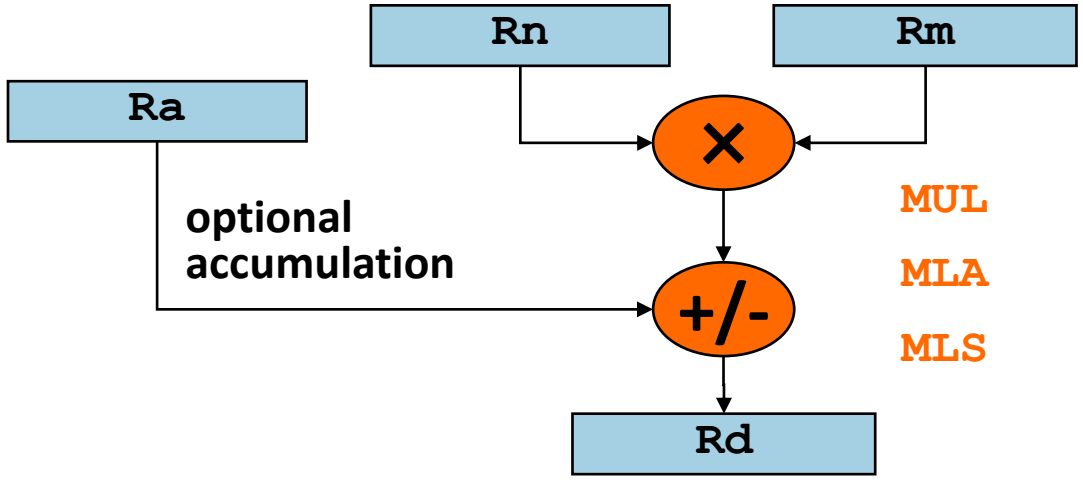
- Data constants loaded into a register through a pair of instructions
- Data side access to instruction memory not required

Literal pools:

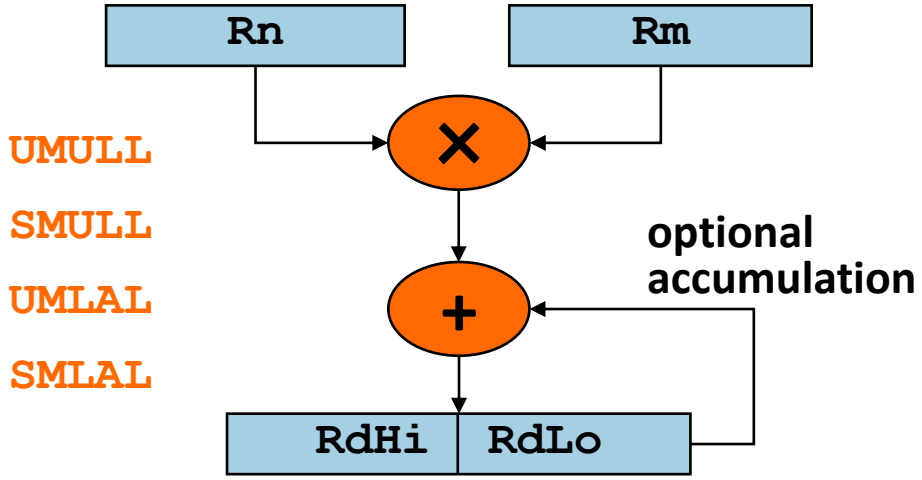
- Data constants, accessed through a PC relative load
- Located near the code that uses them
- Data side access to instruction memory required

Multiply

32-bit multiplication



64-bit multiplication



Examples:

```
int op1, op2, op3, dest1, dest2;
dest1 = op1 * op2;
        MUL r8,r4,r5
dest2 = dest2 + op1 * op2;
        MLA r9,r4,r5,r9
dest1 = dest1 – op2 * op3;
        MLS r8,r5,r6,r8
```

Divide

Armv8-M cores include division hardware

- Signed and unsigned divide are implemented using 32-bit instructions

```
int sdiv(int a, int b)
{
    return (a / b);
}
```

```
sdiv
    SDIV    r0,r0,r1
    BX      lr
```

```
unsigned int udiv(unsigned int c, unsigned int d)
{
    return (c / d);
}
```

```
udiv
    UDIV    r0,r0,r1
    BX      lr
```

Prior to Armv7, Arm cores contained no division hardware

- Division was typically implemented by a run-time library function

Bit manipulation instructions (1)

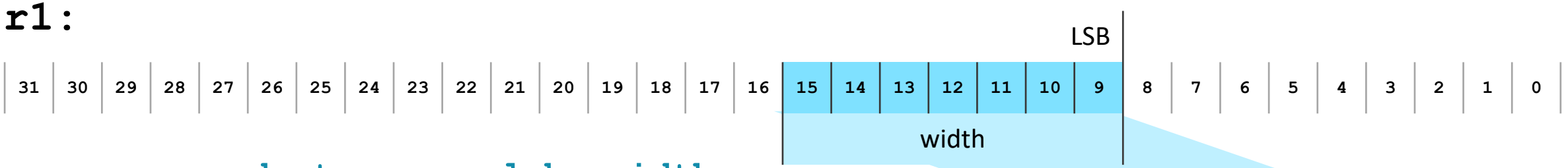
Allow insertion, clearing, extraction and reversal of bits within a register

Typically generated when compiling C code that manipulates bitfields

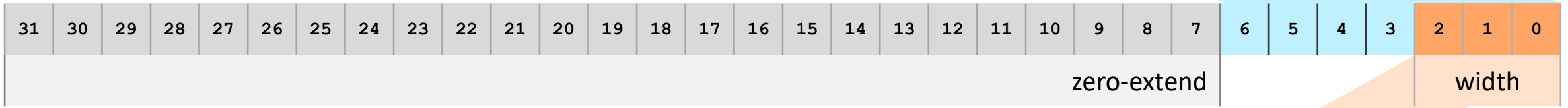
```
struct REG1_t
{
   unsigned int bit_05  :  6;
   unsigned int bit_68  :  3;
   unsigned int bit_9F  :  7;
};

struct REG1_t  RegTemp1;

void test(void)
{
  RegTemp1.bit_68 = RegTemp1.bit_9F;    //  UBFX r2,r1,#9,#7
                                        //  BFI  r0,r2,#6,#3
}
```

Bit manipulation instructions (2)

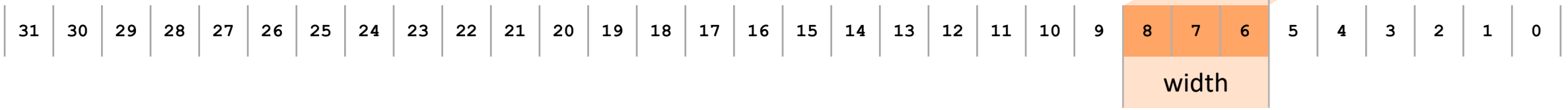
```
r1:
```




```
                dest,  src,   lsb,  width
r2:     UBFX     r2,   r1,    #9,     #7
```




```
                dest,  src,   lsb,  width
r0:     BFI      r0,   r2,    #6,     #3
```



Section quiz - Data processing instructions

```
unsigned int divideByTen(int x)
{
    return x / 10;
}
```

Which of the following is valid disassembly for the C function `divideByTen()`?

```
divideByTen:
    movw    r1, #52429
    movt    r1, #52428
    umull   r0, r1, r0, r1
    lsrs    r0, r1, #3
    bx      lr
```

```
divideByTen:
        udiv    r0, r0, #10
        bx      lr
```

Agenda

Introduction

Data Processing Instructions

Load/Store Instructions

Flow Control

Miscellaneous

Arm Custom Instructions

Learning objectives - Load/store instructions

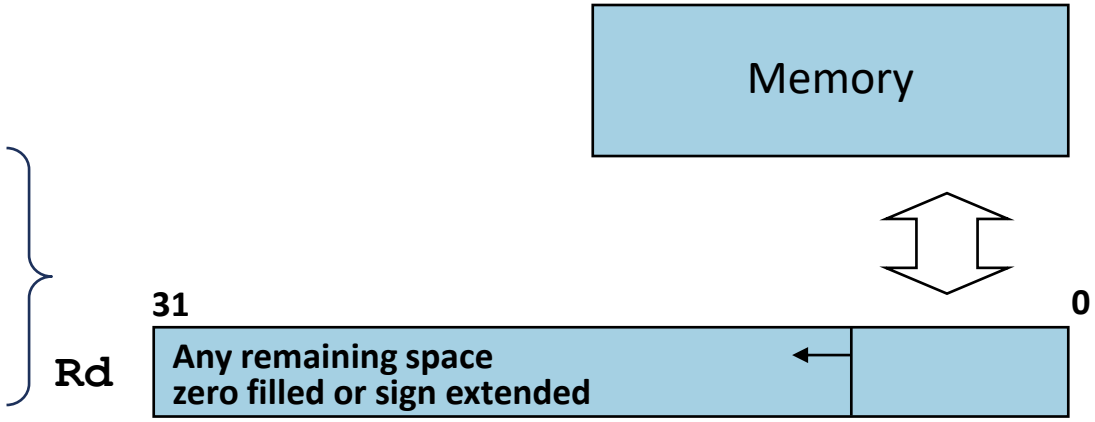
After completing this section, you will be able to:

- Discuss the memory access sizes and addressing modes of load and store instructions in the T32 instruction set

Single/double register data transfer

Use to move data between one or two registers and memory

`LDR` / `STR`	Word
`LDRD` / `STRD`	Doubleword
`LDRB` / `STRB`	Byte
`LDRH` / `STRH`	Halfword
`LDRSB`	Signed byte load
`LDRSH`	Signed halfword load



Syntax

```
LDR{<size>}{<cond>} Rd, <address>
STR{<size>}{<cond>} Rd, <address>
```

Example

```
short *sptr = dest_addr;    // r1
char  *cptr = source_addr;  // r0
*sptr = *cptr;
        LDRB r0,[r0,#0]
        STRH r0,[r1,#0]
```

Addressing memory - Offsets

The address accessed by `LDR` / `STR` is specified by a base register with an optional offset

The offset can be of 4 types

- Base register only (no offset)

```
LDR r0, [r1]
```

- Base register plus constant

```
LDR r0, [r1, #8]
```

- Base register, plus register (optionally shifted by an immediate value)

```
LDR r0, [r1, r2]
LDR r0, [r1, r2, LSL #2]
```

- The offset can be either added to or subtracted from the base register

```
LDR r0, [r1, #-8]
LDR r0, [r1, -r2]
LDR r0, [r1, -r2, LSL #2]
```

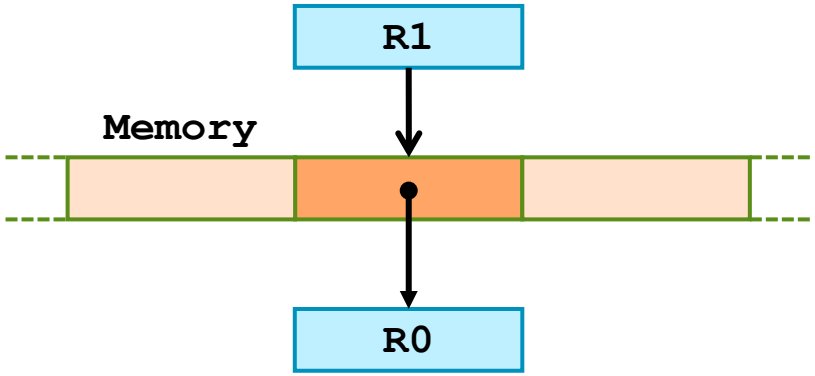
Why is the offset here **#4**?

```
short *sptr = (short *)0x1002;
int op1;
op1 = *(sptr+2);

        LDRSH r1,[r0,#4]
```

Addressing memory - Addressing modes

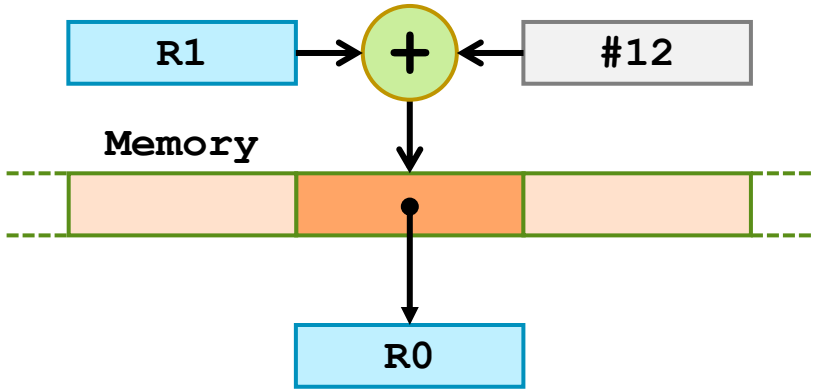
Simple

```
LDR R0, [R1]
```



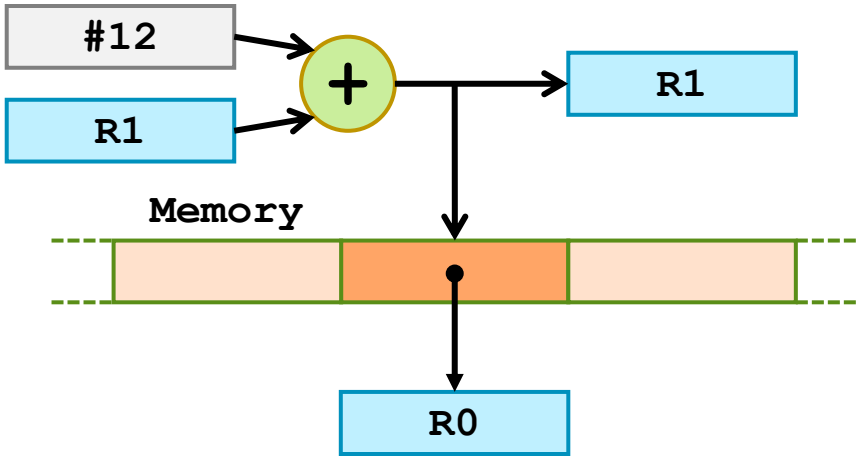
Offset

```
LDR R0, [R1, #12]
```



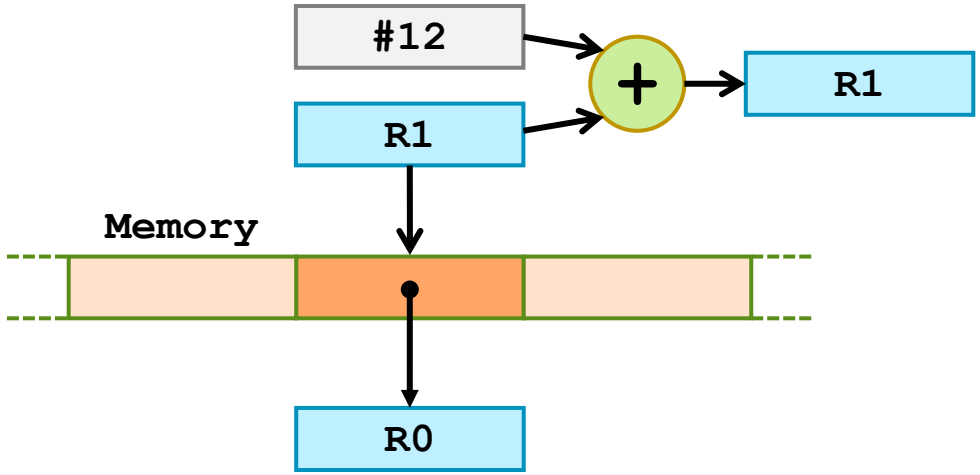
Pre-indexed

```
LDR R0, [R1, #12]!
```



Post-indexed

```
LDR R0, [R1], #12
```



Multiple register data transfer

These instructions move data between multiple registers and memory

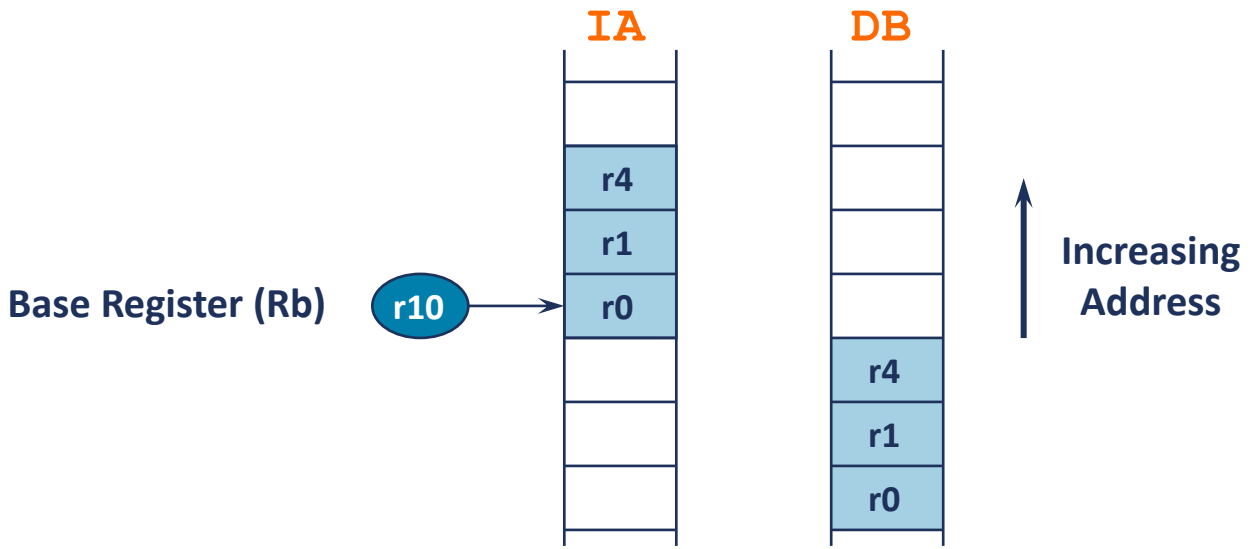
- `<LDM|STM>{<addressing_mode>}{<cond>} Rb{!}, <register list>`

Two addressing modes supported in T32 state

- Increment after (IA)
- Decrement before (DB)
- When used as `LDMIA` and `STMDB`, these modes have the common name Full Descending (FD)

Example

- `LDM    r10, {r0,r1,r4}    ; load registers using r10 base`



Stacks

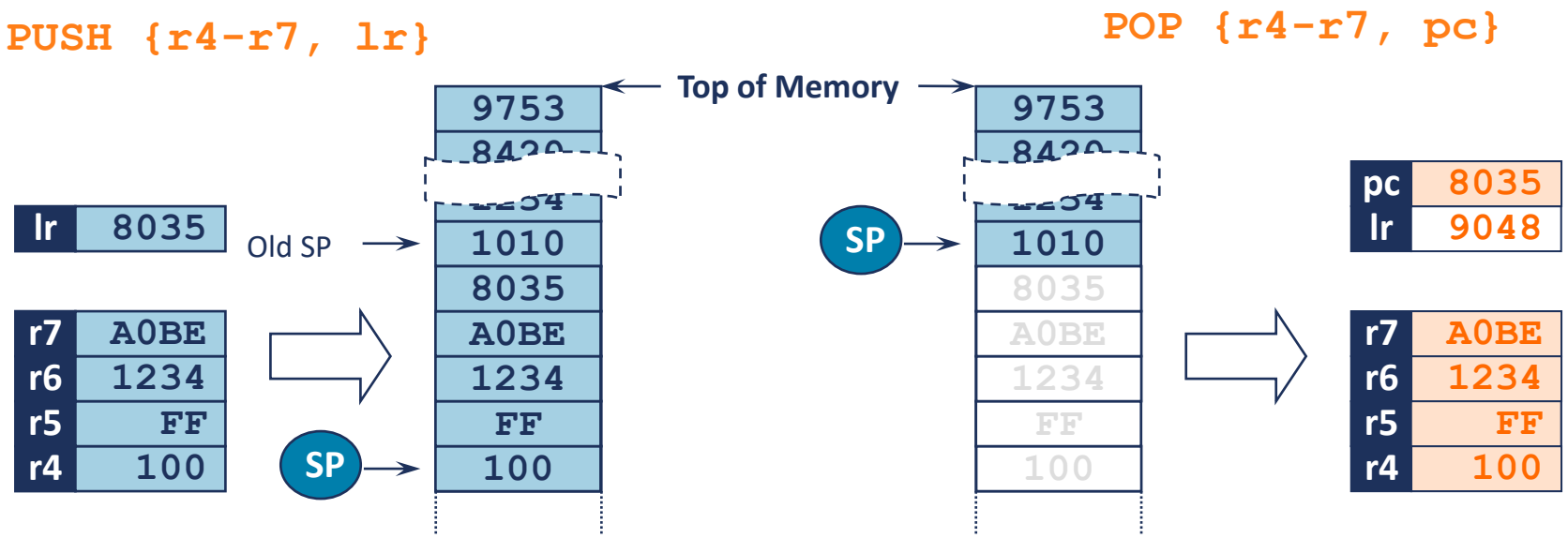
Stack operations are implemented as multiple register transfers

- **PUSH** Store Multiple - Full Descending stack (`STMFD`, `STMDB`)
- **POP** Load Multiple - Full Descending stack (`LDMFD`, `LDMIA`)

Registers are stacked in order from lowest register to lowest memory location

- The order registers are specified has no effect

Stack should normally be kept 8-byte aligned on function entry for AAPCS compliance



Section quiz - Load/store instructions

What is the error in the following instruction sequence that performs a string copy operation?

```
    ldr     r1, =source     // pointer to source
    ldr     r2, =dest       // pointer to destination
copy:
    ldrb    r0, [r1, #1]    // load character from source
    strb    r0, [r2, #1]    // store character to destination
    cmp     r0, #0          // the string is null-terminated
    bne     copy            // copy characters until string ends
    bx      lr
```

Section quiz - Load/store instructions

What is the error in the following instruction sequence that performs a string copy operation?

```
    ldr     r1, =source     // pointer to source
    ldr     r2, =dest       // pointer to destination
copy:
    ldrb    r0, [r1, #1]    // load character from source
    strb    r0, [r2, #1]    // store character to destination
    cmp     r0, #0          // the string is null-terminated
    bne     copy            // copy characters until string ends
    bx      lr
```

Infinite loop!

Section quiz - Load/store instructions

What is the error in the following instruction sequence that performs a string copy operation?

```
        ldr     r1, =source     // pointer to source
        ldr     r2, =dest       // pointer to destination
copy:
        ldrb    r0, [r1, #1]    // load character from source
        strb    r0, [r2, #1]    // store character to destination
        cmp     r0, #0          // the string is null-terminated
        bne     copy            // copy characters until string ends
        bx      lr
```

```
ldrb    r0, [r1, #1]! // increment pointer and load character from source
strb    r0, [r2, #1]! // increment pointer and store character to destination
```

Section quiz - Load/store instructions

What is the error in the following instruction sequence that performs a string copy operation?

```
        ldr     r1, =source     // pointer to source
        ldr     r2, =dest       // pointer to destination
copy:
        ldrb    r0, [r1, #1]    // load character from source
        strb    r0, [r2, #1]    // store character to destination
        cmp     r0, #0          // the string is null-terminated
        bne     copy            // copy characters until string ends
        bx      lr
```

```
ldrb    r0, [r1, #1]! // increment pointer and load character from source
strb    r0, [r2, #1]! // increment pointer and store character to destination
```

Skips first character!

Section quiz - Load/store instructions

What is the error in the following instruction sequence that performs a string copy operation?

```
        ldr     r1, =source     // pointer to source
        ldr     r2, =dest       // pointer to destination
copy:
        ldrb    r0, [r1, #1]    // load character from source
        strb    r0, [r2, #1]    // store character to destination
        cmp     r0, #0          // the string is null-terminated
        bne     copy            // copy characters until string ends
        bx      lr
```

```
ldrb    r0, [r1, #1]! // increment pointer and load character from source
strb    r0, [r2, #1]! // increment pointer and store character to destination
```

```
ldrb    r0, [r1], #1  // load character from source,          and then increment pointer
strb    r0, [r2], #1  // and store character to destination, and then increment pointer
```

Agenda

Introduction

Data Processing Instructions

Load/Store Instructions

Flow Control

Miscellaneous

Arm Custom Instructions

Learning objectives - Flow control

After completing this section, you will be able to:

- Describe the program flow of known a T32 instruction sequence.

Flow control summary

Branch instructions vary in size and range

Instruction	Branch Range	
	16-bit	32-bit
B	+/- 2KB	+/- 16MB
B<cond>	-256 to +254 bytes	+/- 1MB
BL		+/- 16MB
BX	Any[1]	
BXNS	Any[2]	
BLX	Any[1]	
BLXNS	Any[2]	
CBZ	+4 to 130 bytes	
CBNZ	+4 to 130 bytes	
TBB		+512 bytes
TBH		+128KB

[1] Register-based address anywhere in 4GB address space

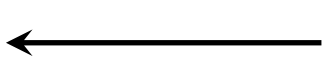
[2] Available if Security Extension is present (must be executed from Secure state)

Branch instructions

Branch instructions have the following format

- `B{<cond>} label`
- Branch range depends on instruction set and width
- Assembler works out the offset of the label from the PC for the branch instruction

```
        B    start
        MOVS r0, r1
lab1
        ADDS r0, #1
        ..
        ..
start
        CMP r0, r1
        ..
```



perform PC-relative branch to label “start”

continue execution from here

Subroutines: Branch with Link

Implementing a conventional subroutine call requires two steps

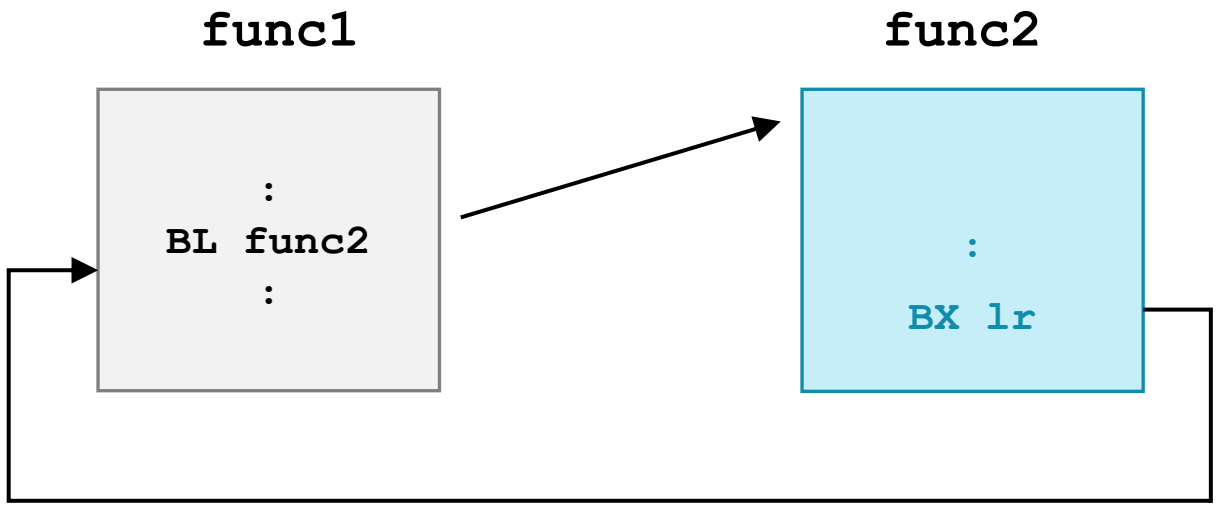
- Store the return address
- Branch to the address of the required subroutine

These steps are carried out in one instruction, `BL`

- The return address is stored in the link register (`lr/r14`)
- Branch to an address (range dependent on instruction set and width)

Returning is performed by restoring the program counter (`pc`) from `lr`

```
void func1(void)
{
        :
        func2();
        :
}
```



Compare and Branch on Zero

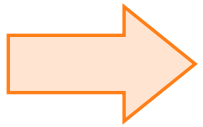
Replaces a CMP followed by a Branch

- BUT does not affect condition code flags
- Can only branch forward between 4 and 130 bytes

Syntax

- `CB{N}Z <Rn>, <label>`
  - `CBZ:` If Rn is equal to zero, branch to label
  - `CBNZ:` If Rn is not equal to zero, branch to label

```
    .
    CMP r0, #0
    BEQ exit
    .
exit
```



```
    .
    CBZ r0, exit
    .
    .
exit
```

If-Then block

Not enough bits in 16-bit or 32-bit Thumb encoding for conditional execution

So `IT` instruction added, along with IT bits in xPSR

Makes the next 1-4 instructions conditional

Syntax

- `IT{T|E}{T|E}{T|E} <condition_code>`
- Any condition code may be used
- Condition flags can change inside the block

Current "if-then status" stored in EPSR

- Conditional block may be safely interrupted
- Not recommended to branch into or out of 'if-then' block

```
; if (r0 == 0)
;   r0 = *r1 + 2;
; else
;   r0 = *r2 + 4;


; if
  CMP r0, #0
  I T T E E    EQ

; then
  LDREQ r0, [r1]
  ADDEQ r0, #2

; else
  LDRNE r0, [r2]
  ADDNE r0, #4
```

Section quiz - Flow control

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                 mov              r0, #2        // array index
                 ldr              r1, =array    // pointer to array
                 cmp              r0, #1
                 blt              L0
                 beq              L1
                 bgt              L2
L0:
                 ldr              r0, [r1]
                 b                exit
L1:
                 ldr              r0, [r1, #4]
                 b                exit
L2:
                 ldr              r0, [r1, #8]
                 b                exit
exit:
                 bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2       // array index
                  ldr              r1, =array   // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

Section quiz - Flow control

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  cmp              r0, #1
                  blt              L0
                  beq              L1
                  bgt              L2
L0:
                  ldr              r0, [r1]
                  b                exit
L1:
                  ldr              r0, [r1, #4]
                  b                exit
L2:
                  ldr              r0, [r1, #8]
                  b                exit
exit:
                  bx     lr
```

At the end of this instruction sequence, what is the value in register `r0`?

What C syntax could this code correspond do?

Can you think of a more optimal way of performing this operation?

Section quiz - Flow control

At the end of this instruction sequence, what is the value in register `r0`?

What C syntax could this code correspond do?

Can you think of a more optimal way of performing this operation?

```
array    dcd              0xF0000000, 0xF0000001, 0xF0000002

lookup:
                  mov              r0, #2        // array index
                  ldr              r1, =array    // pointer to array
                  ldr              r0, [r1, r0, lsl #2]
                  bx               lr
```

Agenda

Introduction

Data Processing Instructions

Load/Store Instructions

Flow Control

Miscellaneous

Arm Custom Instructions

Learning objectives - Miscellaneous

After completing this section, you will be able to:

- Identify whether an assembly language source file has been written using legacy **`armasm`** syntax or using GNU assembler syntax
- Write a T32 instruction sequence to modify an Armv8-M special-purpose register value

Example assembly file

Arm assemblers like armclang and armasm comply to UAL

- The names and use of T32 instructions are unchanged

However expressions and directives vary across different assemblers

armasm

```
      AREA |.text|, CODE, READONLY
      ENTRY
abc   EQU   54
foo
      MOVS   r0, #10
      MOVS   r1, #abc
      ADDS   r2, r0, r1    ; this is a comment
      ...
      DCD   0xAB00321A
      END
```

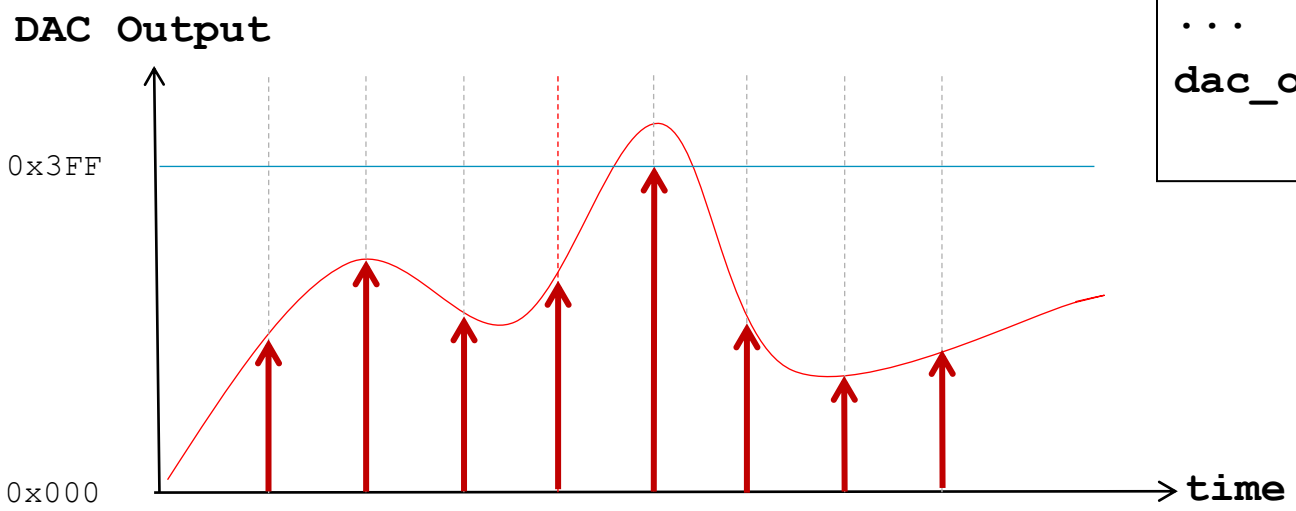
armclang

```
      .section .text, "ax"
      // armclang provides no equivalent to ENTRY
      .equ abc, 54
foo:
      MOVS   r0, #10
      MOVS   r1, #abc
      ADDS   r2, r0, r1        // this is a comment
      ...
      .word 0xAB00321A
      .end
```

Saturation

Saturate a value to a specified bit position (a power of 2)

- Unsigned saturation of 32-bit value: `USAT{<cond>} Rd, #sat, Rm {,shift}`
  - `#sat` is specified as an immediate value in the range 0 to 31
  - `{shift}` is optional and is limited to 5-bit LSL or ASR
  - Q flag is set if saturation occurs (sticky bit)



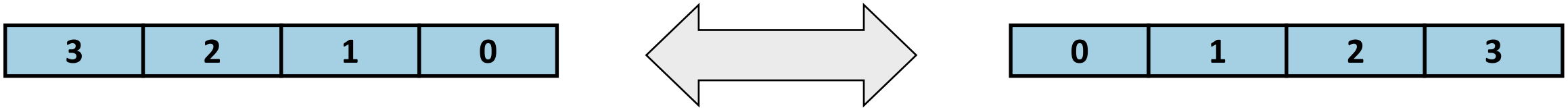

```
unsigned int dac_val;, dac_out;
...
dac_out = __usat(dac_val,10);
        USAT r1,#10,r0
```

- Signed saturation of 32-bit value: `SSAT Rd, #sat, Rm {shift}`
  - Use C intrinsic: `int __SSAT(int val, unsigned int sat)`

Byte reversal and CLZ

Byte Reversal Instructions

```
dest3 = __REV(dest3);
        REV r4,r4
```

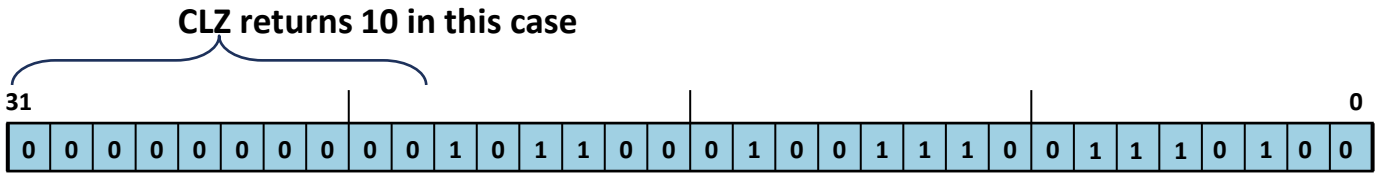


- `REV16{cond} Rd, Rm` Reverses the bytes in each halfword
- `REVSH{cond} Rd, Rm` Reverses the bottom two bytes, and sign extends the result to 32 bits

Count Leading Zeros

- Returns number of unset bits before the most significant set bit

```
dest2 = __CLZ(dest3);
        CLZ r1,r4
```



Accessing special-purpose registers

The MRS/MSR instructions can be used to access special-purpose registers

- `MRS Rd, <reg>` `; Moves from special-purpose registers to Rd`
- `MSR <reg>, Rm` `; Moves from Rm to special-purpose registers`

Able to access all internal registers

- Stack pointers (MSP, PSP)
- Status registers (IPSR, EPSR, APSR, IEPSR, IAPSR, EAPSR, PSR)
- Interrupt registers (PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK)
- CONTROL register

Rules and restrictions

- Thread mode cannot read alternate stack pointer or IPSR – zeros will be returned
- All EPSR bits read as zero during normal execution but can be read when in halting debug mode
- The APSR can be written to by software in Handler or Thread mode
- Software is not permitted to write to the IPSR or EPSR
- BASEPRI_MAX option updates the BASEPRI mask register only when the new value increases the priority level

Power management instructions

Note that these instructions are “hint” instructions

- They may not be implemented on an actual core

Wait For Interrupt – WFI

- Puts the core into standby mode
- Woken by an interrupt or debug event

```
__WFI();
        WFI
```

Wait For Event – WFE

- Same as WFI, but will also wake the core on a signalled event

```
__WFE();
        WFE
```

Send Event – SEV

- Signals an event to other cores in a multi-processor system

```
__SEV();
        SEV
```

No operation

No Operation – NOP

- Can be used as padding to align following instructions
- May or may not take time to execute

```
__NOP();
        NOP
```

Other miscellaneous instructions

There are many more instructions available in Armv8-M some of which are covered in other modules:

- Access permissions and security state information instructions – `TT`, `TTA`, `TTT`, `TTAT`
- Atomics, e.g., `LDA` and `STL`
- Breakpoint instruction – `BKPT`
- Barrier instructions – `DSB`, `DMB` and `ISB`
- DSP and Floating-Point instructions, e.g. `UASX` and `VADD`
- Load/store exclusive instructions, e.g., `LDREX` and `STREX`
- Security extension instructions – `BXNS`, `BLXNS`, `SG`
- Supervisor Call – `SVC`

Section quiz - Miscellaneous

What instruction sequence can be used to switch from the Main Stack Pointer (MSP) to the Process Stack Pointer (PSP)?

Section quiz - Miscellaneous

What instruction sequence can be used to switch from the Main Stack Pointer (MSP) to the Process Stack Pointer (PSP) for Thread mode?

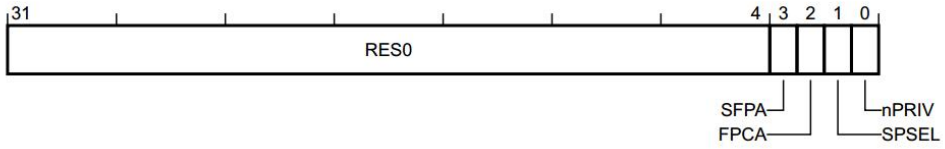
D1.2.13 CONTROL, Control Register

The CONTROL characteristics are:

Purpose	Provides access to the PE control fields.
Usage constraints	Privileged access only, but unprivileged writes are ignored unless otherwise specified.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is banked between Security states on a bit by bit basis.

Field descriptions

The CONTROL bit assignments are:



Section quiz - Miscellaneous

What instruction sequence can be used to switch from the Main Stack Pointer (MSP) to the Process Stack Pointer (PSP) for Thread mode?

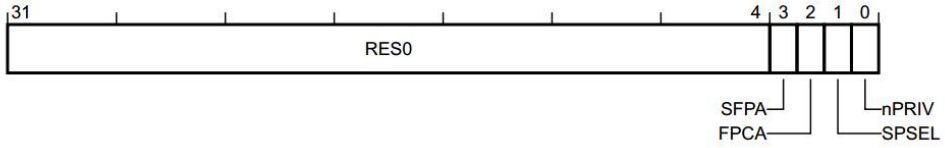
D1.2.13 CONTROL, Control Register

The CONTROL characteristics are:

Purpose	Provides access to the PE control fields.
Usage constraints	Privileged access only, but unprivileged writes are ignored unless otherwise specified.
Configurations	This register is always implemented.
Attributes	32-bit read/write special-purpose register. This register is banked between Security states on a bit by bit basis.

Field descriptions

The CONTROL bit assignments are:



SPSEL, bit [1]

Stack-pointer select. Defines the stack pointer to be used.

This bit is banked between Security states.

The possible values of this bit are:

0	Use SP_main as the current stack.
1	In Thread mode use SP_process as the current stack.

This bit resets to zero on a Warm reset.

Section quiz - Miscellaneous

What instruction sequence can be used to switch from the Main Stack Pointer (MSP) to the Process Stack Pointer (PSP) for Thread mode?

```
        .section .text, "ax"

        .global switch_to_psp
        .type switch_to_psp, %function

switch_to_psp:
    mrs r0, CONTROL  // read the CONTROL register into r0
    orr r0, r0, #0x2 // switch to the Process Stack Pointer
    msr CONTROL, r0  // write r0 out to the CONTROL register
    bx lr
```

Agenda

Introduction

Data Processing Instructions

Load/Store Instructions

Flow Control

Miscellaneous

Arm Custom Instructions

Learning objectives – Arm Custom Instructions (ACIs)

After completing this section, you will be able to:

- Interpret the instruction patterns of ACIs

Custom instruction classes

Three classes of instruction

```
<operation code> <destination reg>, <imm>
<operation code> <dest reg>, <source reg>, <imm>
<operation code> <dest reg>, <src reg 1>, <src reg 2>, <imm>
```

Source and destination registers can be either:

- **General purpose registers (and `APSR_nzcv` condition codes)**
- **FP and SIMD registers (no condition codes)**

Immediate values can also be encoded

Note: Cortex-M33 does not support using the SIMD variant

Coprocessor and accumulation

Custom instructions use the same encoding space as the coprocessor instructions

- Each instruction targets a specific coprocessor

Instructions can optionally accumulate into the destination register

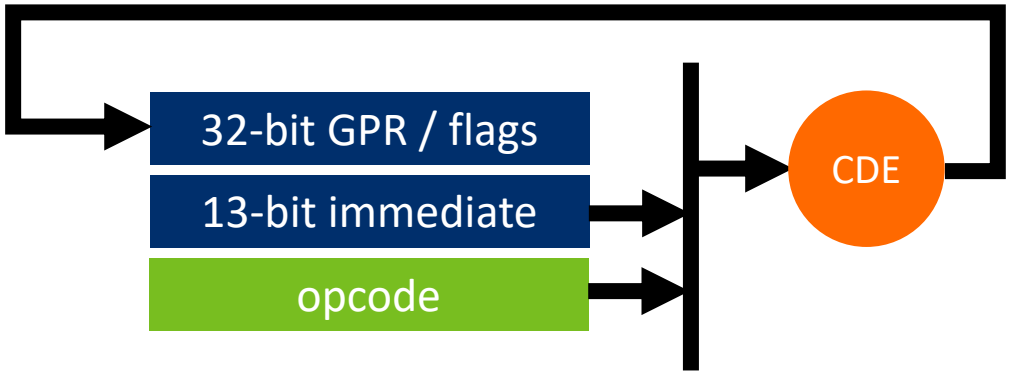
- Only accumulating versions of integer class of instructions can be inside an IT Block

The integer class instructions have a “dual” variant, whose result is 64-bit

General-purpose register file – Custom instruction class 1

```
CX1 p0, APSR_nzcv, #3
```

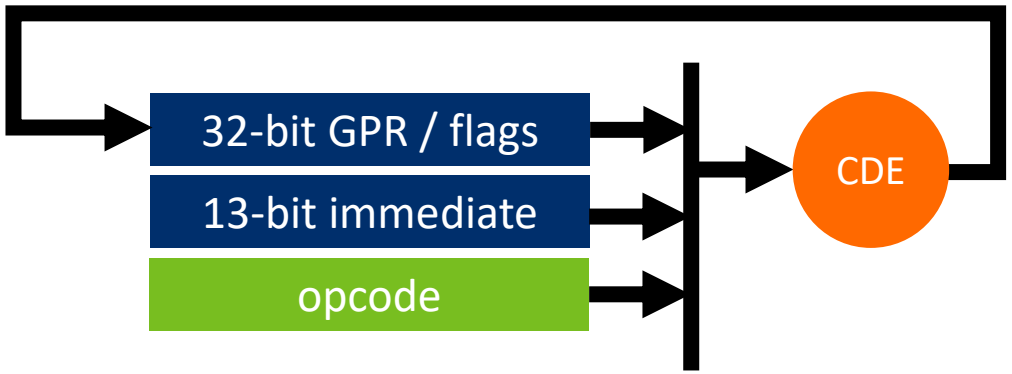
Non-accumulator variant



Operate using condition flags

```
CX1A p1, r2, #4
```

Accumulator variant

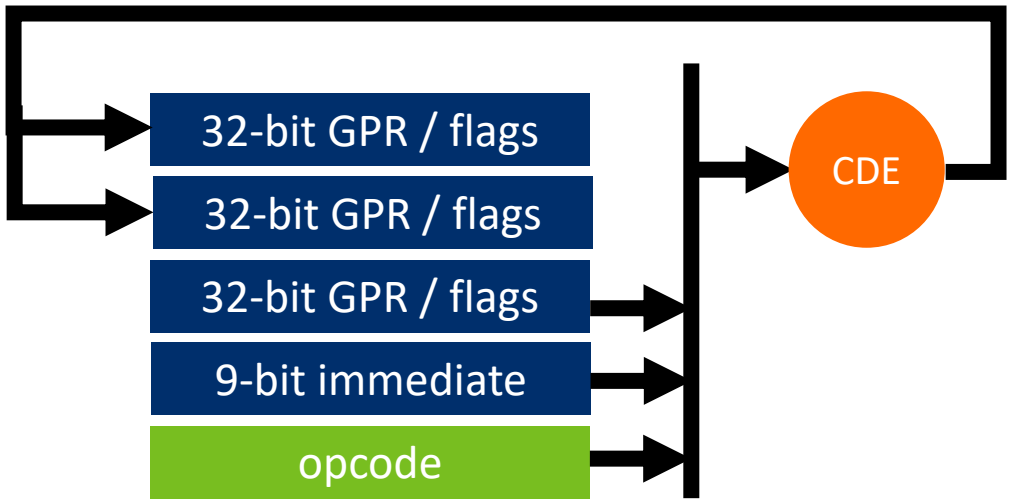


Accumulate into r2

General-purpose register file – Custom instruction class 2

```
CX2D p1, r0, r1, r5, #6
```

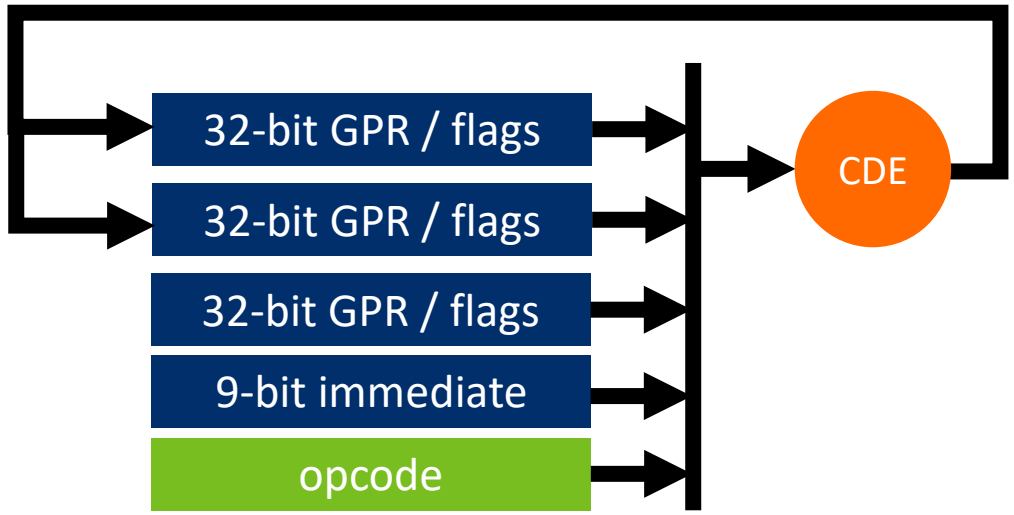
Non-accumulator variant



Two destination registers

```
CX2DA p3, r3, r4, r8, #0xab
```

Accumulator variant



Two destination registers and accumulation

Thank You
Danke
Gracias
Grazie
谢谢
ありがとう
Asante
Merci
감사합니다
धन्यवाद
Kiitos
شكرًا
ধন্যবাদ
תודה