Module 1-B

ISA Overview

Tim Rogers 2022

Reading

- Textbook, Chapter 3, “ARM Instruction Set Architecture”, pages 55 – 74.
  - We’re using an ARM Cortex-M0 (ARMv6) for this class. Not an STM32L4 with an ARM Cortex-M4 (ARMv7) as described by your textbook.
- **<u>Textbook, Chapter 4, “Arithmetic and Logic”, pages 75 – 96.</u>**
- ARMv6-M Architecture Reference Manual (436 pages)
  - Get familiar with sections A5.2 (16-bit Thumb encoding) and A6.7 (Alphabetical list of ARMv6-M Thumb instructions)

Learning Outcome #1

"An ability to program a microcontroller to perform various tasks"

How?

✓ A. Architecture and Programming Model
B. Instruction Set Basics
C. Addressing modes and conditionals
D. Practical Assembly Language Programming
E. Stacks and Recursive Functions*

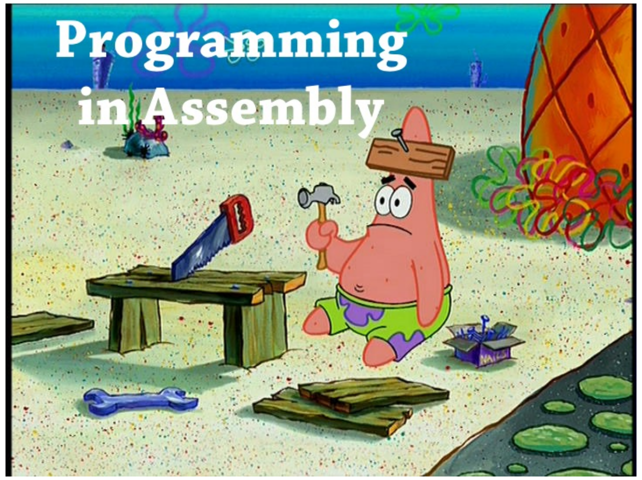
Objective

“Instruction Set Overview”

Why?

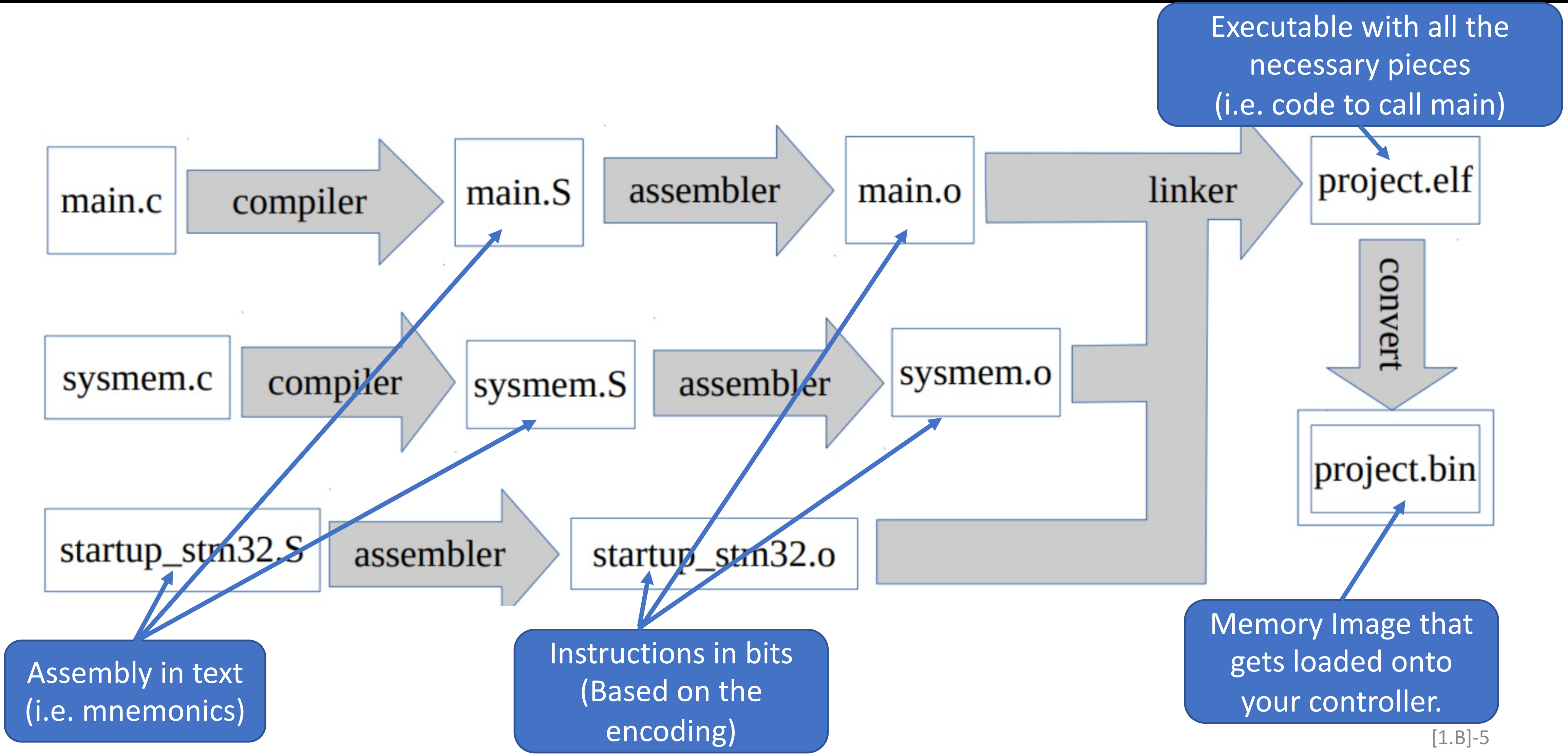
Prof. Rogers’ Perspective

362 Student’s Perspective?



- **Actual ECE 362 Meme (Author unknown)**

Where the ISA fits in the flow


Executable with all the necessary pieces (i.e. code to call main)
main.c
compiler
main.S
assembler
main.o
linker
project.elf
sysmem.c
compiler
sysmem.S
assembler
sysmem.o
convert
startup_stm32.S
assembler
startup_stm32.o
project.bin
Assembly in text (i.e. mnemonics)
Instructions in bits (Based on the encoding)
Memory Image that gets loaded onto your controller.

Overview of the Cortex-M0 ISA

ARM Cortex-M0 CPUs use only the "Thumb" instruction set.

- The Thumb ISA is not the ARM ISA.

The Cortex-M0 has only 56 different instructions.

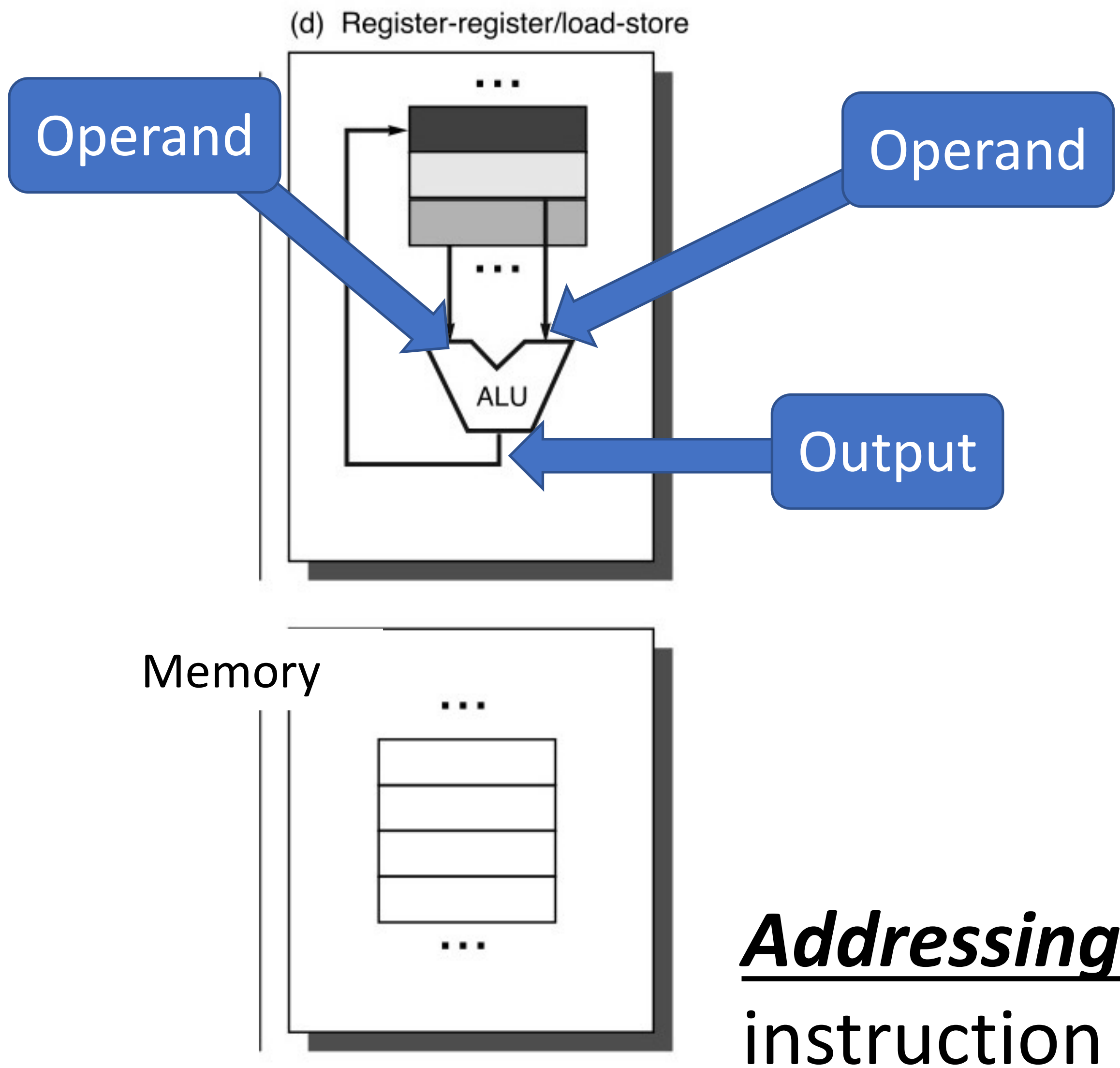
- There are 50 Thumb instructions that are each 16 bits (2 bytes) long.
- There are 6 Thumb2 instructions that are each 32 bits (4 bytes) long.
  - Actually they are 16-bit instructions that tell the CPU there is one more 16-bit chunk.
  - There is only one 32-bit instruction that we ever need to use. (The "BL" instruction)

0-, 1-, 2-, and 3-operand instructions.

Standard operand order for our assembler:

- e.g. adds r5,r2,r4 means r5 = r2 + r4
- e.g. muls r3,r1 means r3 = r3 * r1

Addressing Mode and Operands



__Addressing mode__ determines where instruction inputs/outputs come from / go to

Recall: Our addressing modes

- **Immediate**: small value contained in instruction
- **Register**: src/dst register in instruction
- **Offset**: value is at an address that's a relative (constant) offset from a register
- **Indexed**: value is at address specified by register plus a second register

Immediate Addressing

In the binary representation of "**ADDS R6, R2, #5**" (the top box), which bits encode R2?

A: [1:0]
B: [2:0]
C: [9:8]
D: [5:3]
E: [4:3]

A6.7.2 ADD (immediate)

This instruction adds an immediate value to a register value, and writes
It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

```
ADDS <Rd>,<Rn>,#<imm3>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

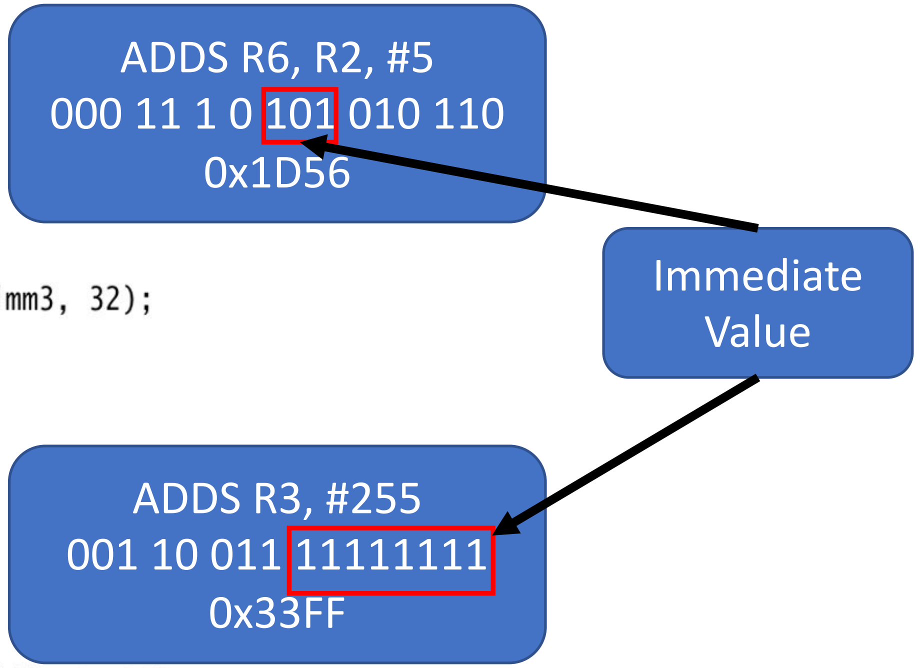
```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb instruction set.

```
ADDS <Rdn>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```



A6.7.2 ADD (immediate)

This instruction adds an immediate value to a register value, and writes the result to the destination register. It updates the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

`ADDS <Rd>,<Rn>,#<imm3>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	1	0	imm3			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

Encoding T2 All versions of the Thumb instruction set.

`ADDS <Rdn>,#<imm8>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	Rdn			imm8							

```
d = UInt(Rdn);  n = UInt(Rdn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);
```

Assembler syntax

`ADDS{<q>} {<Rd>,} <Rn>, #<const>` All encodings permitted

where:

S	The instruction updates the flags.
{<q>}	See *Standard assembler syntax fields* on page A6-98.
<Rd>	The destination register. If <Rd> is omitted, this register is the same as <Rn>.
<Rn>	The register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus immediate)* on page A6-111. If the PC is specified for <Rn>, see *ADR* on page A6-115.
<const>	The immediate value to be added to the value obtained from <Rn>. The range of permitted values is 0-7 for encoding T1, and 0-255 for encoding T2. Encoding T1 is preferred to encoding T2 if <Rd> is specified and encoding T2 is preferred to encoding T1 if <Rd> is omitted.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        APSR.V = overflow;
```

Exceptions

None.

A6.3 Conditional execution

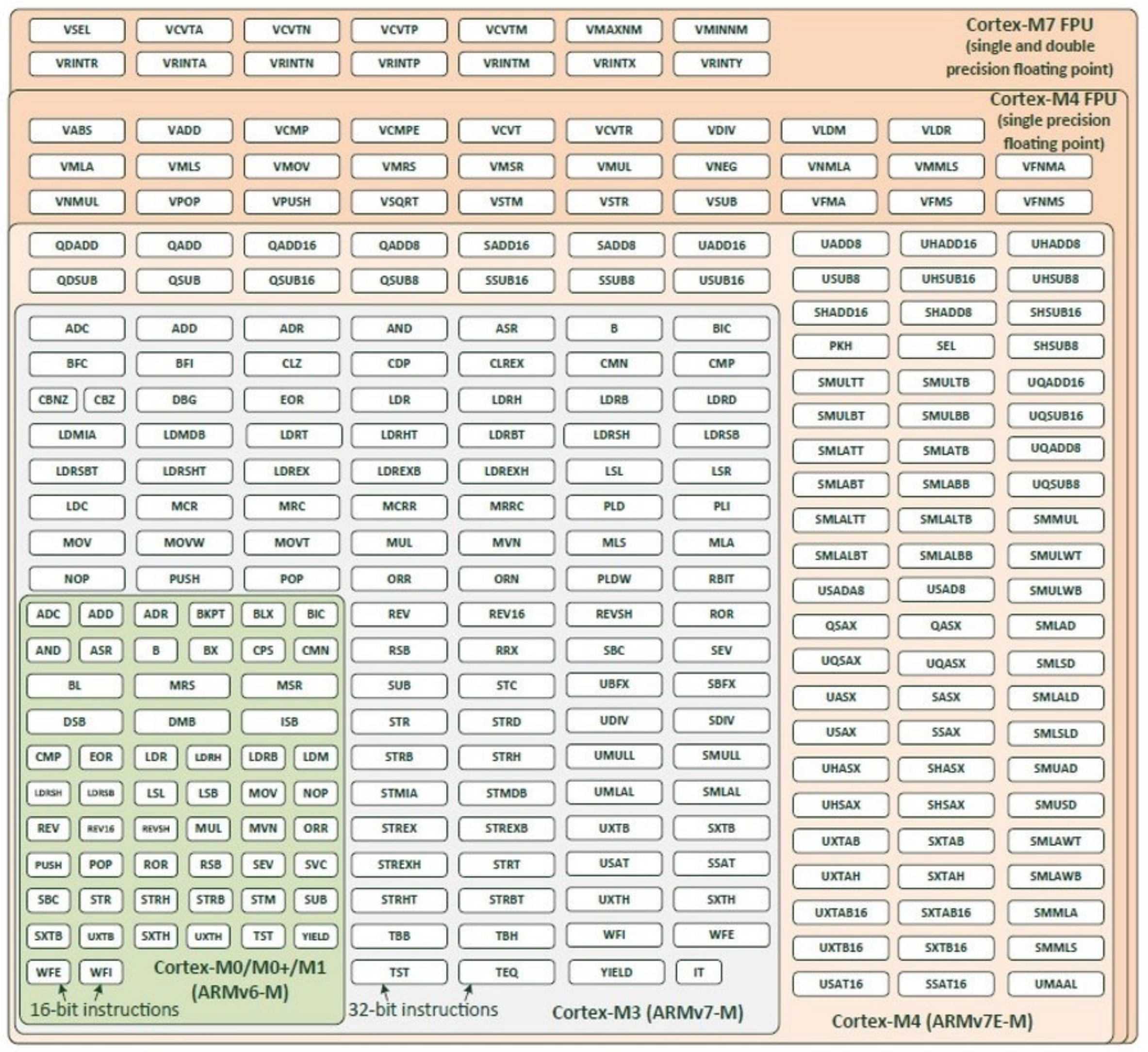
In Thumb instructions, the condition, if it is not AL, is normally encoded in a preceding IT instruction. However, ARMv6-M does not support the IT instruction. This means that:

- the <c> suffix must be omitted or AL in all instruction mnemonics except B<c>
- in the pseudocode in this manual:
  - any reference to InITBlock() returns FALSE
  - any reference to LastInITBlock() returns FALSE.

Why?
“Unified Assembly Language”
However – you can pretty much ignore all of it for Cortex-M0

Cortex ISAs

- There are many flavors of ARM Cortex-M CPUs.
  - We're using M0.
- But unified assembly language covers all of them.
- And so does your book.



[1.B]-11

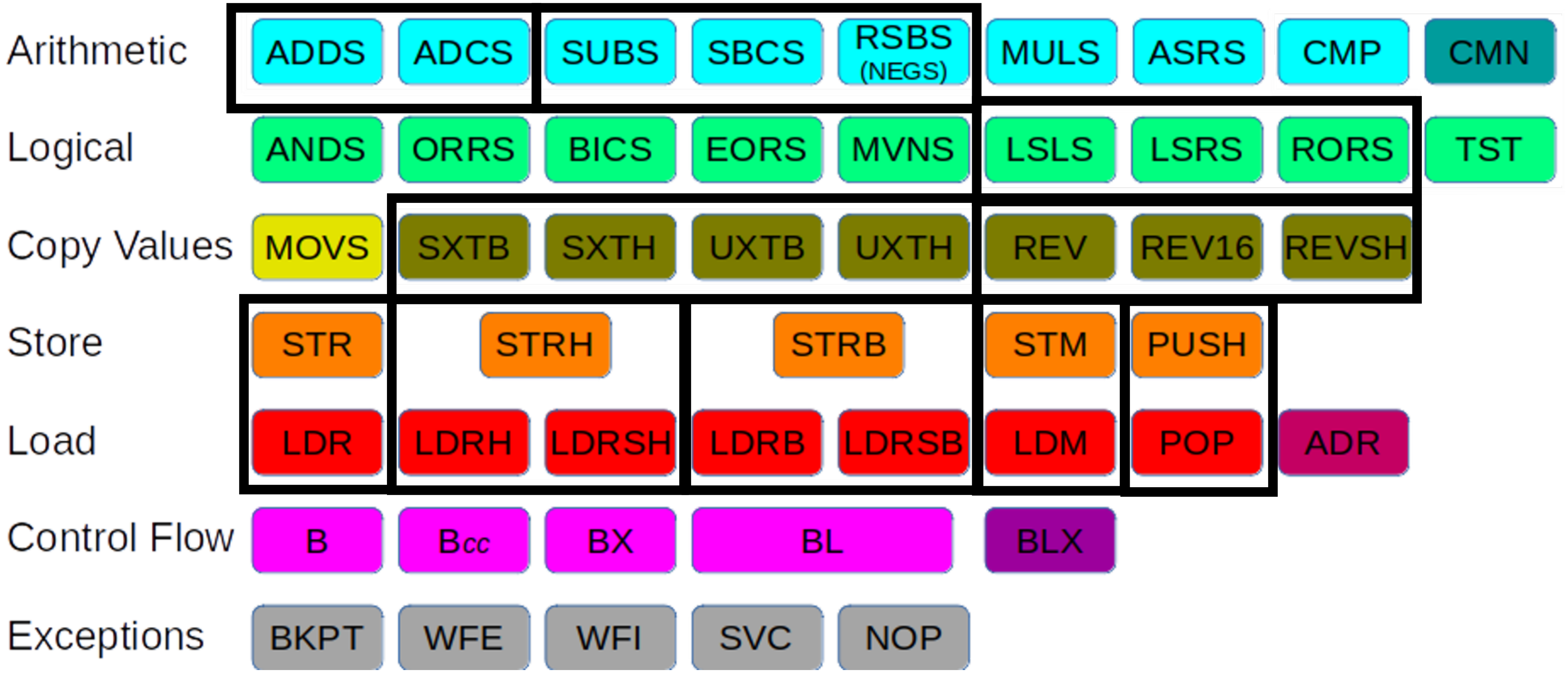
If in doubt, trust the instruction encoding

If you read a description that sounds more complicated than it needs to be, look at the instruction encoding.

If there’s no way to encode the functionality talked about in the "Operation" section, you can probably ignore it.

This will probably take some time to get used to. That’s why we have lots of practice.

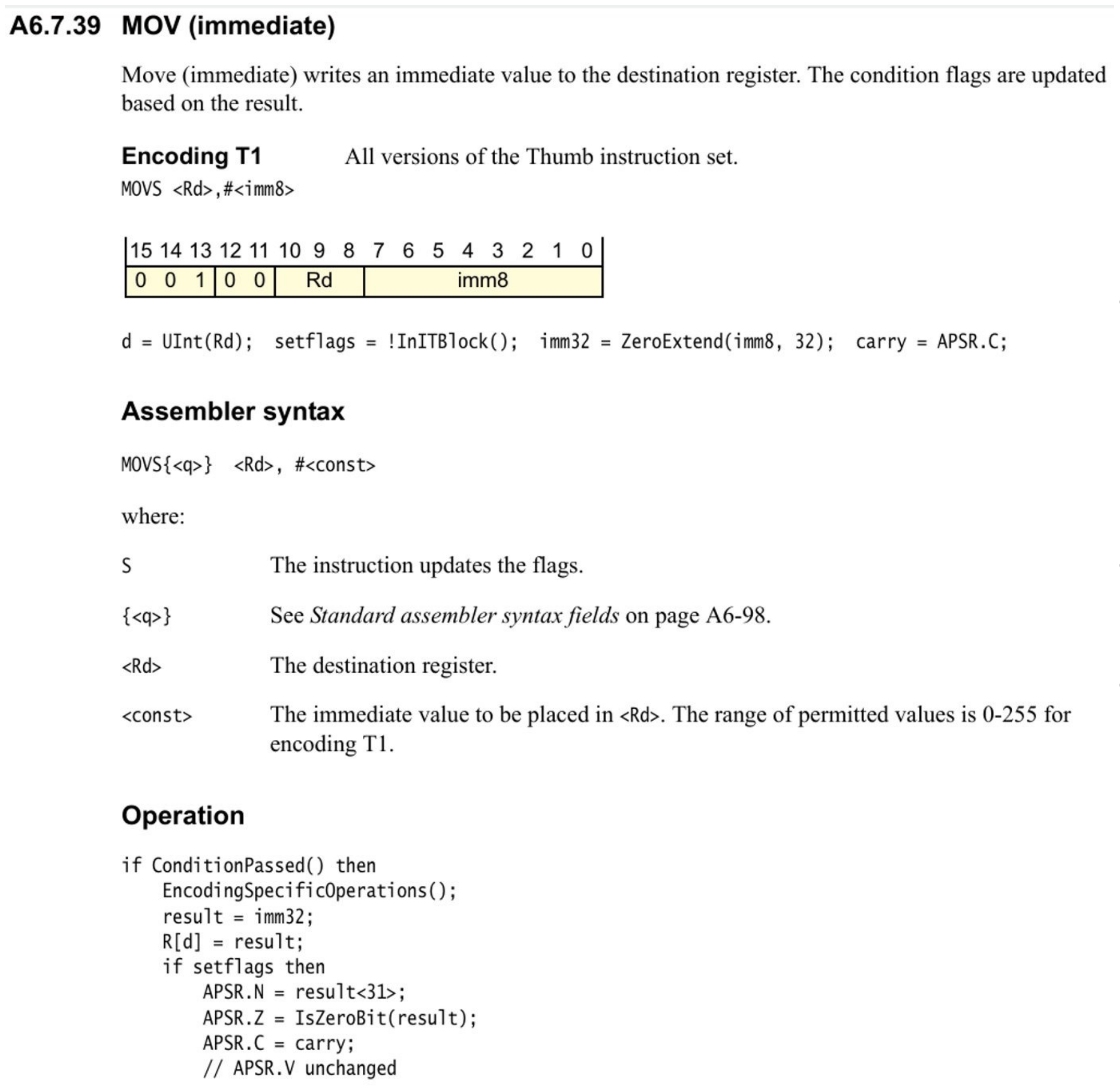
Rick’s Handy Cortex-M0 Summary



Arithmetic	ADDS	ADCS	SUBS	SBCS	RSBS (NEGS)	MULS	ASRS	CMP	CMN
Logical	ANDS	ORRS	BICS	EORS	MVNS	LSLS	LSRS	RORS	TST
Copy Values	MOVS	SXTB	SXTH	UXTB	UXTH	REV	REV16	REVSH	
Store	STR	STRH		STRB		STM	PUSH		
Load	LDR	LDRH	LDRSH	LDRB	LDRSB	LDM	POP	ADR	
Control Flow	B	Bcc	BX	BL		BLX			
Exceptions	BKPT	WFE	WFI	SVC	NOP				

Back to Immediate operands These instructions allow immediates:

- ADDS: Add a number
- ASRS: Arithmetic Shift Right by a number of bits
- CMP: Compare to a number
- LSLS: Logical Shift Left by a number of bits
- LSRS: Logical Shift Right by a number of bits
- MOVS: Move immediate number into a register
- RSBS: Reverse Subtract a number (result = #imm – Rn) (only for #0!)
- SUBS: Subtract a number (result = Rn - #imm)



A6.7.39 MOV (immediate)

Move (immediate) writes an immediate value to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb instruction set.

```
MOVS <Rd>,#<imm8>
```

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	Rd			imm8							

```
d = UInt(Rd);  setflags = !InITBlock();  imm32 = ZeroExtend(imm8, 32);  carry = APSR.C;
```

Assembler syntax

```
MOVS{<q>}  <Rd>, #<const>
```

where:

S	The instruction updates the flags.
{<q>}	See *Standard assembler syntax fields* on page A6-98.
<Rd>	The destination register.
<const>	The immediate value to be placed in <Rd>. The range of permitted values is 0-255 for encoding T1.

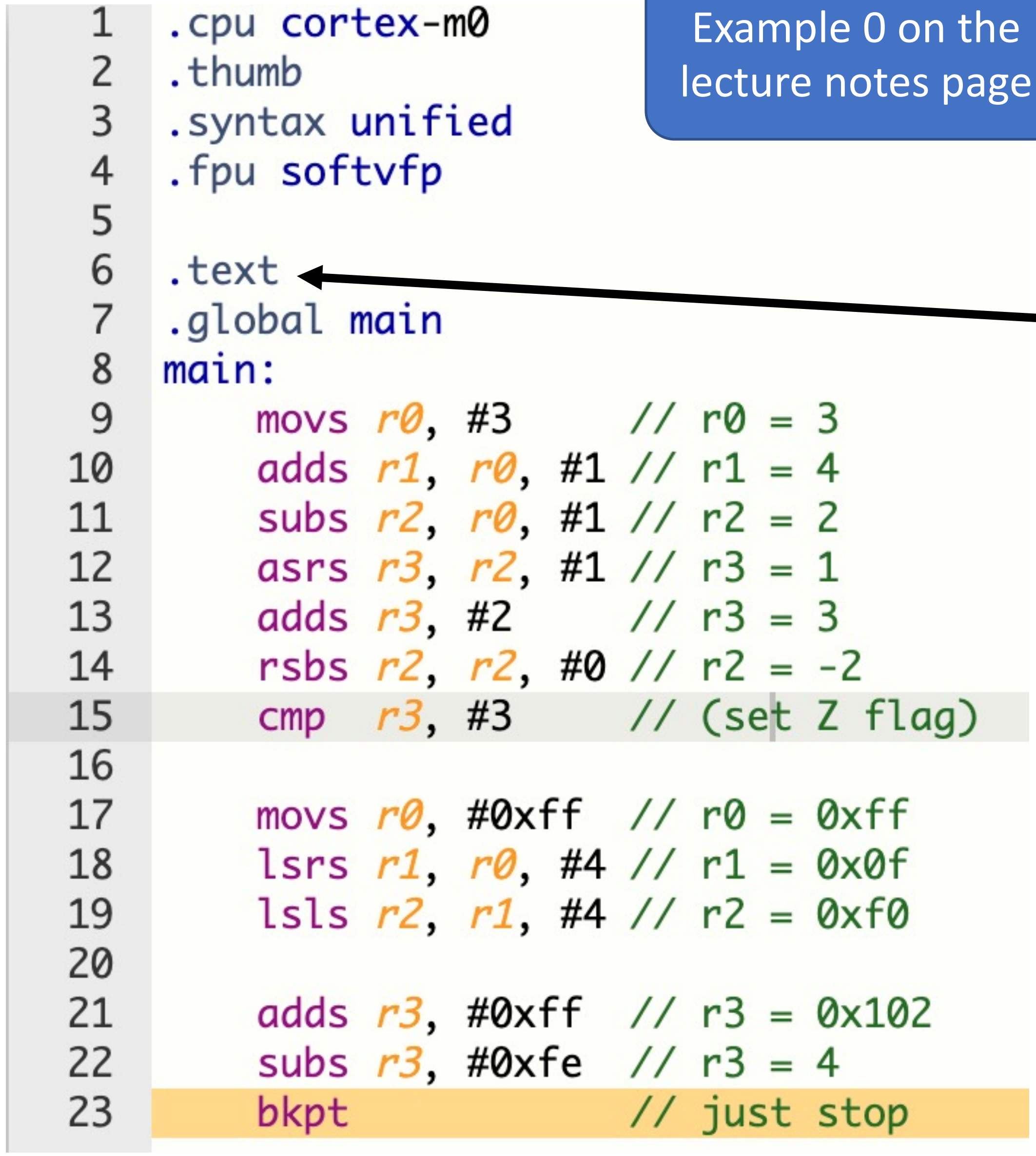
Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    result = imm32;
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

MOV (immediate) instruction

- Looks almost like ADDS Rdn, #imm8.
  - The opcode is different.
  - Sets the flags, but neither Carry nor overflow.
  - Can only reference regs 0-7
- How about ConditionPassed()?
  - Ignore it.
- What about S?
  - No way to select whether or not the MOV (immediate) instruction updates the APSR flags.
  - Need to always specify MOVS when using immediate operand.

Code Example with Immediate Operands



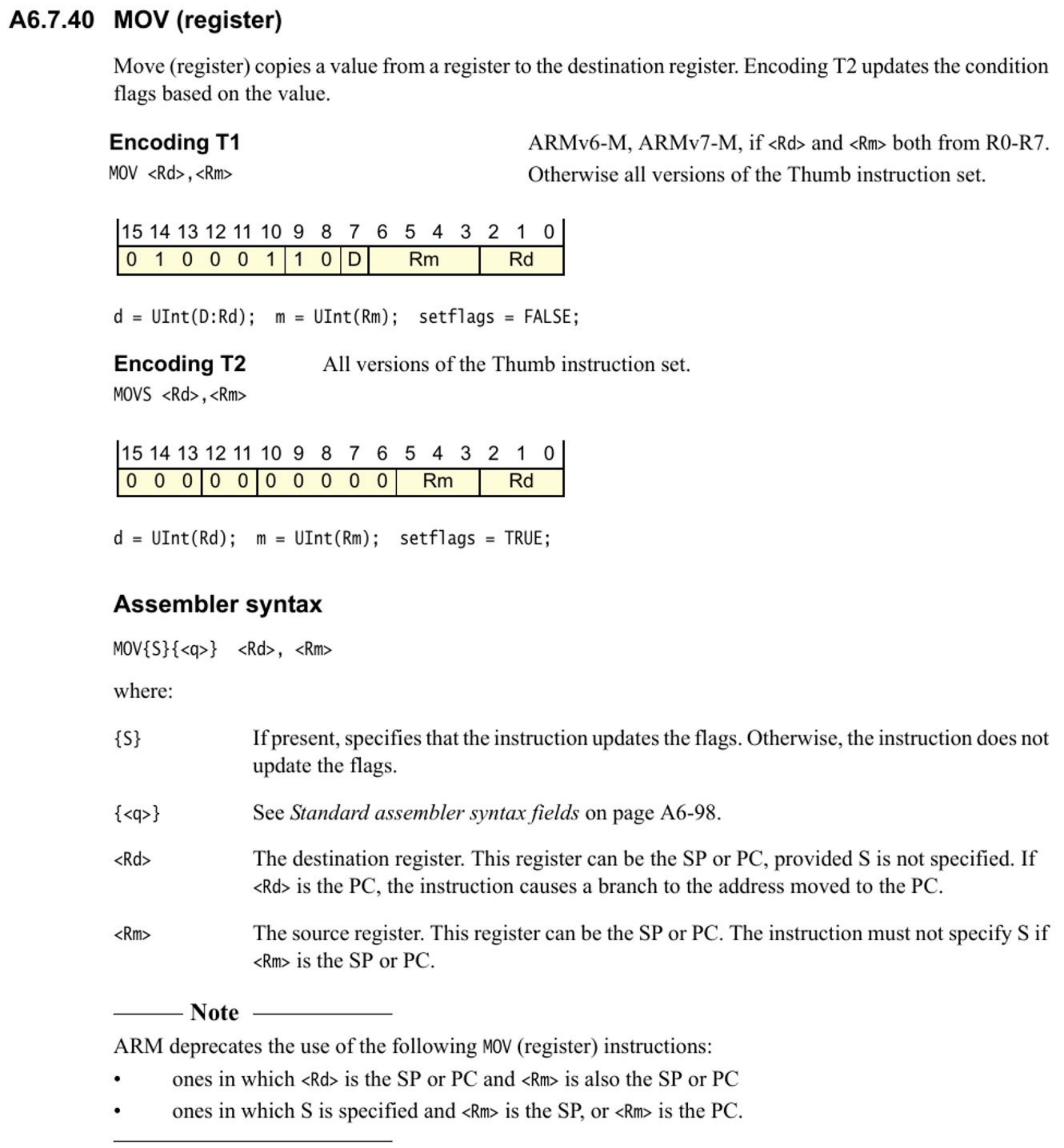
```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:
    movs r0, #3     // r0 = 3
    adds r1, r0, #1 // r1 = 4
    subs r2, r0, #1 // r2 = 2
    asrs r3, r2, #1 // r3 = 1
    adds r3, #2     // r3 = 3
    rsbs r2, r2, #0 // r2 = -2
    cmp  r3, #3     // (set Z flag)

    movs r0, #0xff  // r0 = 0xff
    lsrs r1, r0, #4 // r1 = 0x0f
    lsls r2, r1, #4 // r2 = 0xf0

    adds r3, #0xff  // r3 = 0x102
    subs r3, #0xfe  // r3 = 4
    bkpt            // just stop
```

Example 0 on the lecture notes page

Tells the assembler the following is read-only.

Recall: Our addressing modes

- ✓ **Immediate**: small value contained in instruction
- **Register**: src/dst register in instruction
- **Offset**: value is at an address that's a relative (constant) offset from a register
- **Indexed**: value is at address specified by register plus a second register



A6.7.40 MOV (register)

Move (register) copies a value from a register to the destination register. Encoding T2 updates the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M, if <Rd> and <Rm> both from R0-R7. Otherwise all versions of the Thumb instruction set.

`MOV <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
```

Encoding T2 All versions of the Thumb instruction set.

`MOVS <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
```

Assembler syntax

`MOV{S}{<q>}  <Rd>, <Rm>`

where:

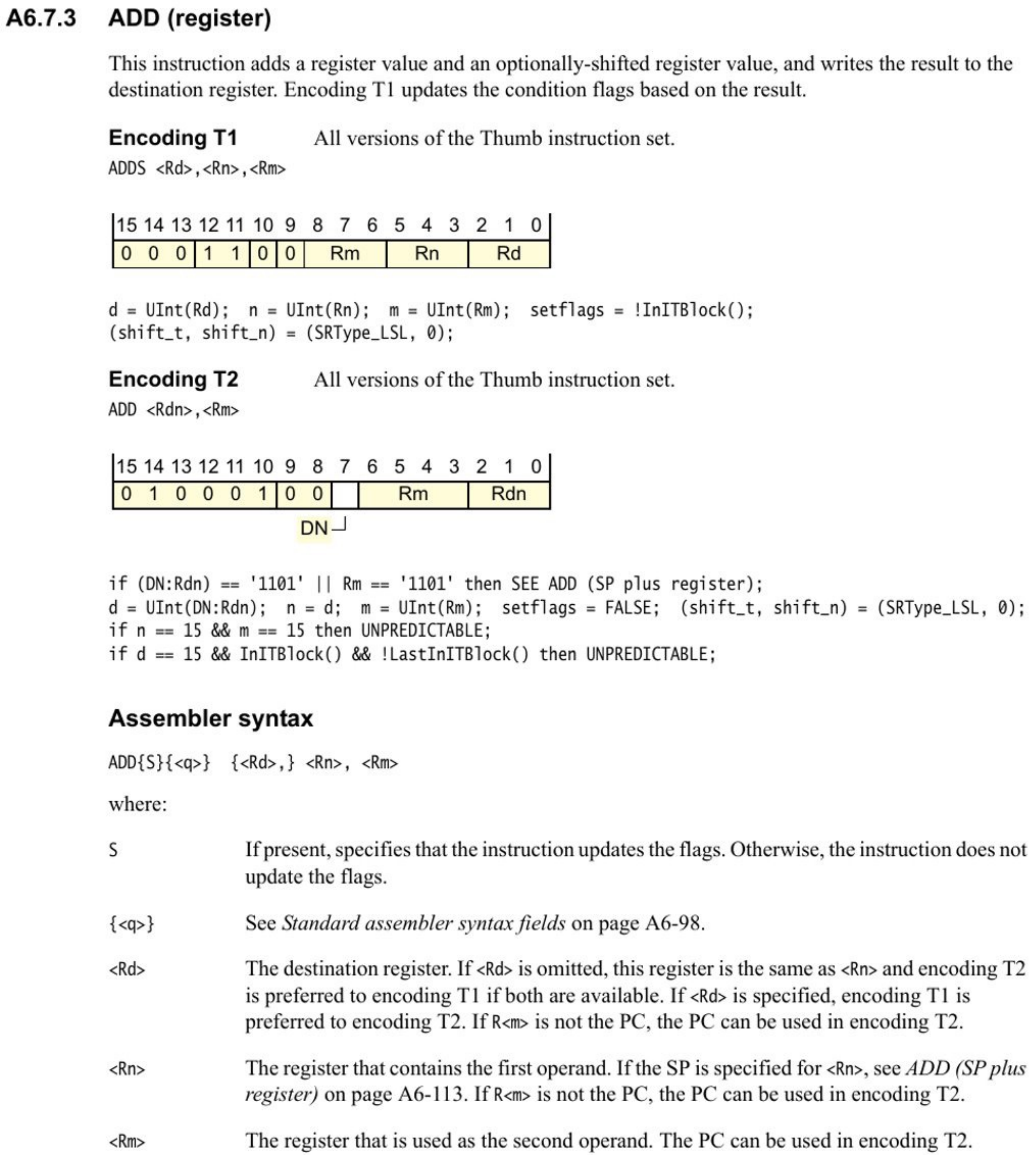
{S}	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
{<q>}	See *Standard assembler syntax fields* on page A6-98.
<Rd>	The destination register. This register can be the SP or PC, provided S is not specified. If <Rd> is the PC, the instruction causes a branch to the address moved to the PC.
<Rm>	The source register. This register can be the SP or PC. The instruction must not specify S if <Rm> is the SP or PC.

Note

ARM deprecates the use of the following MOV (register) instructions:

- ones in which <Rd> is the SP or PC and <Rm> is also the SP or PC
- ones in which S is specified and <Rm> is the SP, or <Rm> is the PC.

- Let's look at MOV (register).
  - You can use R8-R15!
  - You need to be careful though.
  - And it doesn't change the flags.
  - One operand is split.
- The other MOVS is pretty normal.



A6.7.3 ADD (register)

This instruction adds a register value and an optionally-shifted register value, and writes the result to the destination register. Encoding T1 updates the condition flags based on the result.

Encoding T1 All versions of the Thumb instruction set.

`ADDS <Rd>,<Rn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	0	Rm			Rn			Rd		

```
d = UInt(Rd);  n = UInt(Rn);  m = UInt(Rm);  setflags = !InITBlock();
(shift_t, shift_n) = (SRType_LSL, 0);
```

Encoding T2 All versions of the Thumb instruction set.

`ADD <Rdn>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	0	0	DN	Rm				Rdn		

```
if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);
d = UInt(DN:Rdn);  n = d;  m = UInt(Rm);  setflags = FALSE;  (shift_t, shift_n) = (SRType_LSL, 0);
if n == 15 && m == 15 then UNPREDICTABLE;
if d == 15 && InITBlock() && !LastInITBlock() then UNPREDICTABLE;
```

Assembler syntax

`ADD{S}{<q>}  {<Rd>,} <Rn>, <Rm>`

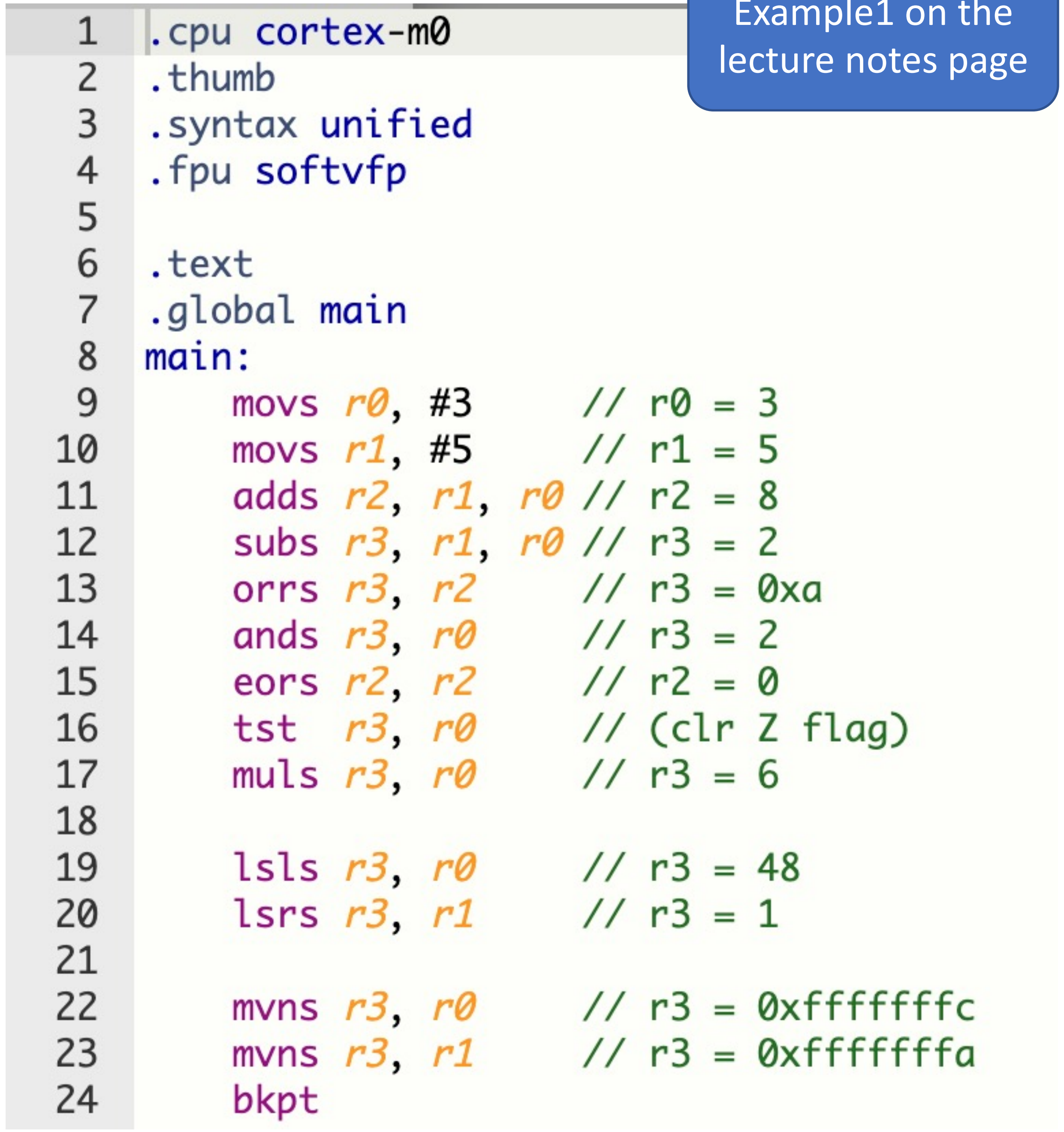
where:

S	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
{<q>}	See *Standard assembler syntax fields* on page A6-98.
<Rd>	The destination register. If <Rd> is omitted, this register is the same as <Rn> and encoding T2 is preferred to encoding T1 if both are available. If <Rd> is specified, encoding T1 is preferred to encoding T2. If R<m> is not the PC, the PC can be used in encoding T2.
<Rn>	The register that contains the first operand. If the SP is specified for <Rn>, see *ADD (SP plus register)* on page A6-113. If R<m> is not the PC, the PC can be used in encoding T2.
<Rm>	The register that is used as the second operand. The PC can be used in encoding T2.

- Other instructions can also refer to all 16 registers
  - ADD
  - CMP
  - MOV

ADCS

- Add with Carry
  - Uses APSR.C as the carry-in (ADD does not)
  - Good for multi-word math
  - 32-registers make this less useful for us



```
.cpu cortex-m0
.thumb
.syntax unified
.fpu softvfp

.text
.global main
main:
    movs r0, #3     // r0 = 3
    movs r1, #5     // r1 = 5
    adds r2, r1, r0 // r2 = 8
    subs r3, r1, r0 // r3 = 2
    orrs r3, r2     // r3 = 0xa
    ands r3, r0     // r3 = 2
    eors r2, r2     // r2 = 0
    tst  r3, r0     // (clr Z flag)
    muls r3, r0     // r3 = 6

    lsls r3, r0     // r3 = 48
    lsrs r3, r1     // r3 = 1

    mvns r3, r0     // r3 = 0xfffffffc
    mvns r3, r1     // r3 = 0xfffffffa
    bkpt
```

Example1 on the lecture notes page

Register operands example

Common Pitfalls

- There are many mistakes that students make repeatedly, so it is worthwhile warning you:
  - There is no "muls r0,r1,r2" instruction. Instead, you must:
    - movs r0,r1
    - muls r0,r2
  - There is no "muls r0,r1,#3" instruction. Instead, you must:
    - movs r0,#3
    - muls r0,r1
  - There is no "ands r1,#7" instruction. Instead, you must:
    - movs r0,#7
    - ands r1,r0

Aside: How to initialize a register with 32-bits

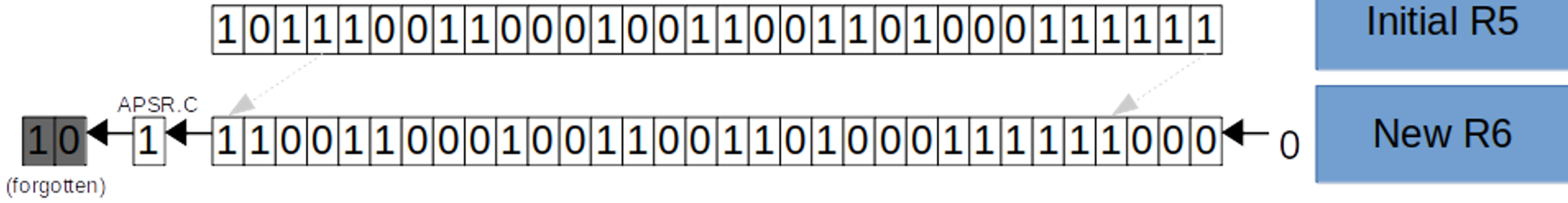
- You won’t understand how this works at this point… You just need to use it.
- We can use an assembler trick to load any value into R0 – R7. For example:
  - LDR R0, =0x12345678
  - Note the "=" sign.

What gets stored in R6?
“**LSLS R6, R5, #0**”
A: 0
B: R5
C: MEM [R5]
D: R6
E: MEM [R6]

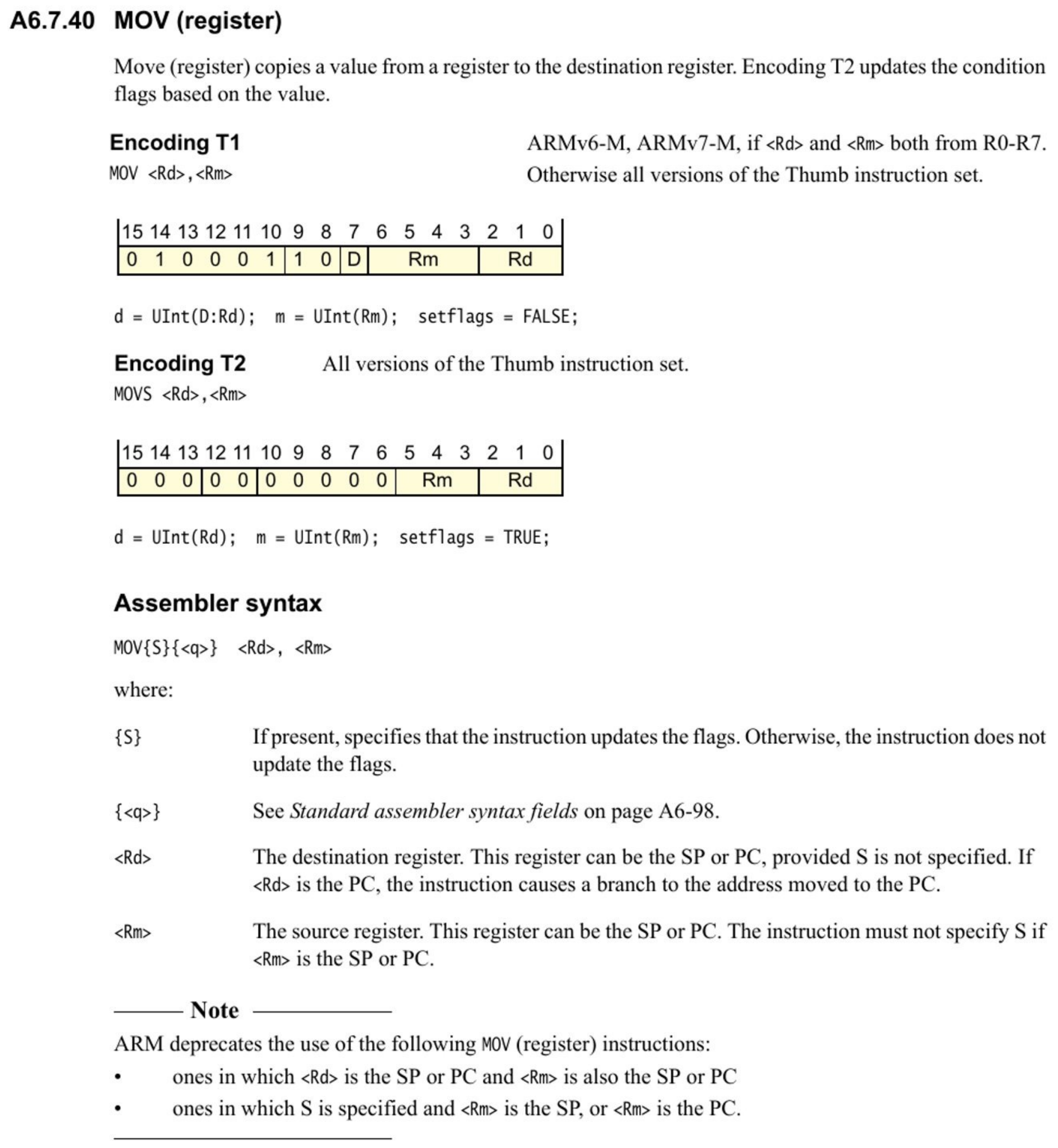
Logical Shift Left

- Can shift the values in 32-bit registers
  - << operator in C
  - Last bit shifted out into APSR.C
  - Zeros shifted in

```
LDR  R5, =0xb9899a3f
LSLS R6, R5, #3
```



Instruction Synonyms



A6.7.40 MOV (register)

Move (register) copies a value from a register to the destination register. Encoding T2 updates the condition flags based on the value.

Encoding T1 ARMv6-M, ARMv7-M, if `<Rd>` and `<Rm>` both from R0-R7.
Otherwise all versions of the Thumb instruction set.

`MOV <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	1	0	0	0	1	1	0	D	Rm				Rd		

```
d = UInt(D:Rd);  m = UInt(Rm);  setflags = FALSE;
```

Encoding T2 All versions of the Thumb instruction set.

`MOVS <Rd>,<Rm>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	Rm			Rd		

```
d = UInt(Rd);  m = UInt(Rm);  setflags = TRUE;
```

Assembler syntax

```
MOV{S}{<q>}  <Rd>, <Rm>
```

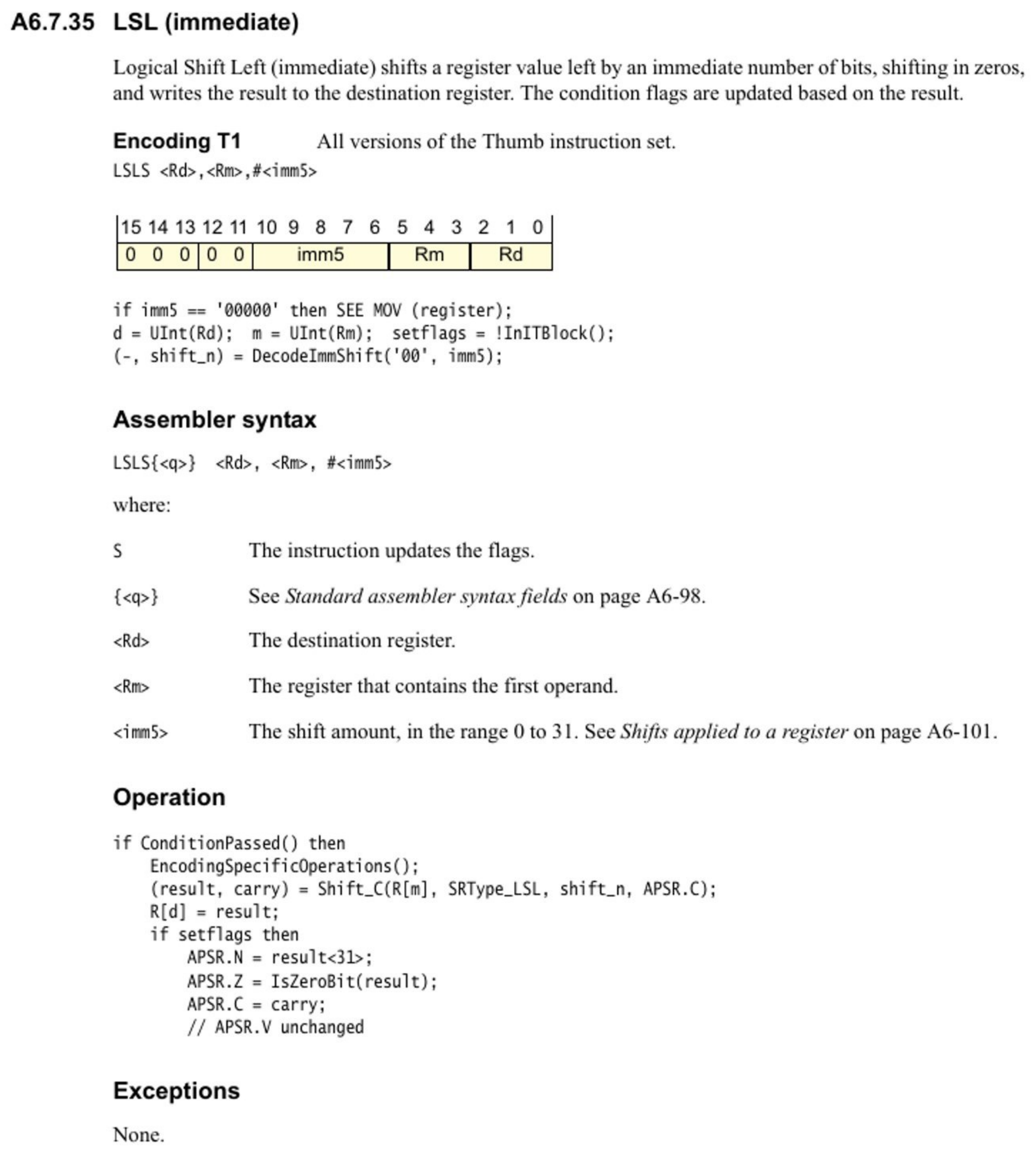
where:

`{S}`	If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.
`{<q>}`	See *Standard assembler syntax fields* on page A6-98.
`<Rd>`	The destination register. This register can be the SP or PC, provided S is not specified. If `<Rd>` is the PC, the instruction causes a branch to the address moved to the PC.
`<Rm>`	The source register. This register can be the SP or PC. The instruction must not specify S if `<Rm>` is the SP or PC.

——— **Note** ———

ARM deprecates the use of the following `MOV` (register) instructions:

- ones in which `<Rd>` is the SP or PC and `<Rm>` is also the SP or PC
- ones in which S is specified and `<Rm>` is the SP, or `<Rm>` is the PC.



A6.7.35 LSL (immediate)

Logical Shift Left (immediate) shifts a register value left by an immediate number of bits, shifting in zeros, and writes the result to the destination register. The condition flags are updated based on the result.

Encoding T1 All versions of the Thumb instruction set.

`LSLS <Rd>,<Rm>,#<imm5>`

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	imm5					Rm			Rd		

```
if imm5 == '00000' then SEE MOV (register);
d = UInt(Rd);  m = UInt(Rm);  setflags = !InITBlock();
(-, shift_n) = DecodeImmShift('00', imm5);
```

Assembler syntax

```
LSLS{<q>}  <Rd>, <Rm>, #<imm5>
```

where:

`S`	The instruction updates the flags.
`{<q>}`	See *Standard assembler syntax fields* on page A6-98.
`<Rd>`	The destination register.
`<Rm>`	The register that contains the first operand.
`<imm5>`	The shift amount, in the range 0 to 31. See *Shifts applied to a register* on page A6-101.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    (result, carry) = Shift_C(R[m], SRType_LSL, shift_n, APSR.C);
    R[d] = result;
    if setflags then
        APSR.N = result<31>;
        APSR.Z = IsZeroBit(result);
        APSR.C = carry;
        // APSR.V unchanged
```

Exceptions

None.