Instruction Sets
Immediate & Register Modes

ECE 362
https://engineering.purdue.edu/ece362/

Reading Assignment

- 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.
- Textbook, Chapter 4, “Arithmetic and Logic”, pages 75 – 96.
- 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)

Instruction Sets for CISC

- A Complex Instruction Set Computer (CISC) normally has a variable-length instruction.
  - e.g. x86-64:
    - RET
      - One byte: c3
    - MOV $8, %AL
      - Two bytes: b0 08
    - MOV $0x1234567890abcdef, %RAX:
      - Ten bytes: 48 b8 ef cd ab 90 78 56 34 12
    - Many ways to combine constants/registers to refer to memory.
      - N = arr[4*x + 16] ==> mov 0x40(%rdi,%rsi,4),%eax ==> 8b 44 b7 40
    - Very space-efficient instructions.

Instruction Sets for RISC

- A Reduced Instruction Set Computer (RISC) has a uniform length instruction.
  - e.g. ARMv7 as you might find in a Raspberry Pi has a 32-bit instruction:
    - MOV r3, $8
      - 4 bytes: e3a00308
    - MOV r0, r5
      - 4 bytes: e1a00005
    - No way to specify large constants.
    - Memory access patterns are limited
    - Not very space-efficient instructions.

Cortex-M0 Instruction Set

- ARM Cortex-M0 CPUs use only the "Thumb" instruction set.
  - The Thumb ISA is not the ARM ISA. It is completely different.
- 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 is right-to-left.
  - e.g. adds r5,r2,r4 means r5 = r2 + r4

Addressing Modes

- The simple computer employed three addressing modes:
  - Immediate: One operand was a register and the other was encoded in the instruction:
    - ADDI R5,#0f (320f)
  - 2-Register: Both operands were registers:
    - ADD R6,R11 (862b)
  - Absolute: One operand referred to a specific 16-bit memory location.
    - LDL R3,01fc (a300 01fc)
- Many CPUs have a way to use absolute addressing. ARM Cortex-M0 does not.
  - Addresses are 32 bits long.
  - Each instruction is always 2 (or 4) bytes long (16 or 32 bits)
  - No room in the instruction for a 32-bit address.

Cortex-M0 Instruction Set

- RISC with compromises
  - Most instructions can refer only to registers R0 – R7.
  - Only small constants are used in instructions.
  - Load/Store architecture: Memory references separate from arithmetic.
- Four major 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

Cortex-M0 Instruction Format

A5.2 16-bit Thumb instruction encoding

The encoding of 16-bit Thumb instructions is:

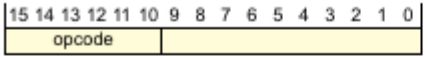


Table A5-1 shows the allocation of 16-bit instruction encodings.

Table A5-1 16-bit Thumb instruction encoding

opcode	Instruction or instruction class
00xxxx	*Shift (immediate), add, subtract, move, and compare* on page A5-85
010000	*Data processing* on page A5-86
010001	*Special data instructions and branch and exchange* on page A5-87
01001x	Load from Literal Pool, see *LDR (literal)* on page A6-141
0101xx 011xxx 100xxx	*Load/store single data item* on page A5-88
10100x	Generate PC-relative address, see *ADR* on page A6-115
10101x	Generate SP-relative address, see *ADD (SP plus immediate)* on page A6-111
1011xx	*Miscellaneous 16-bit instructions* on page A5-89
11000x	Store multiple registers, see *STM, STMIA, STMEA* on page A6-175
11001x	Load multiple registers, see *LDM, LDMIA, LDMFD* on page A6-137
1101xx	*Conditional branch, and Supervisor Call* on page A5-90
11100x	Unconditional Branch, see *B* on page A6-119

ARMv6-M Architecture Reference Manual
Page 84

Top 6 bits of instruction are the “opcode”

Immediate Addressing

- Small integer encoded directly into instruction.
- e.g.

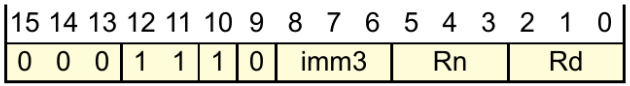
ARMv6 Architecture Reference Manual Page 107

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>
```

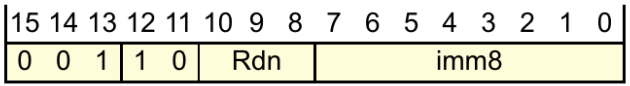



```
d = UInt(Rd);  n = UInt(Rn);  setflags = !InITBlock();  imm32 = ZeroExtend(imm3, 32);
```

ADDS R6, R2, #5
000 11 1 0 101 010 110
0x1D56

Encoding T2 All versions of the Thumb instruction set.

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




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

ADDS R3, #255
001 10 011 11111111
0x33FF

ARMv6 Arch. Ref. Manual p. 107

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);
```

ADDS R6, R2, #5
000 11 1 0 101 010 110
0x1D56

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);
```

ADDS R3, #255
001 10 011 11111111
0x33FF

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.

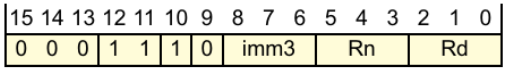
- What happens if I change bit 9 of Encoding T1 for a bigger immediate value?
  - Then it's no longer an ADDS instr.
- Why is the order of values in the opcode not the way it is in the mnemonic?
  - Assembly language is for people. Machine language is for machines.
- What happens to old R3?
  - Gone.
- Why are there two forms of ADDS?
  - It's a trade-off between 3 registers and a short immediate operand OR 2 registers and a longer operand.
- What is {<q>}? !InITBlock()?
  - *sigh*

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.

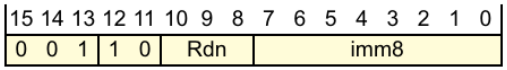
```
ADDS <Rd>,<Rn>,#<imm3>
```




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

Encoding T2 All versions of the Thumb instruction set.

```
ADDS <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

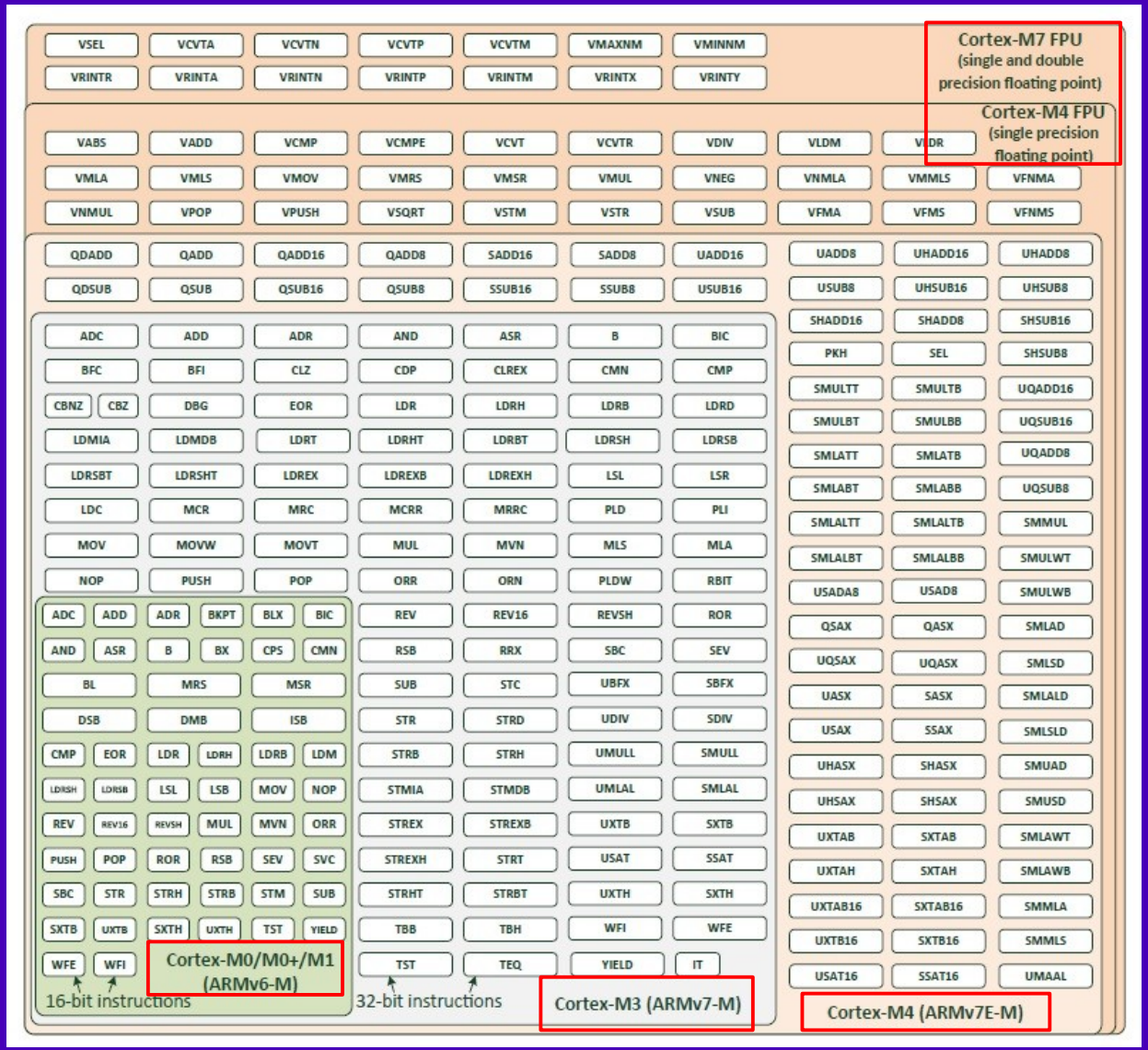
None.

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 is all this stuff here?
For Unified Assembly Language.
You can ignore it all for Cortex-M0!

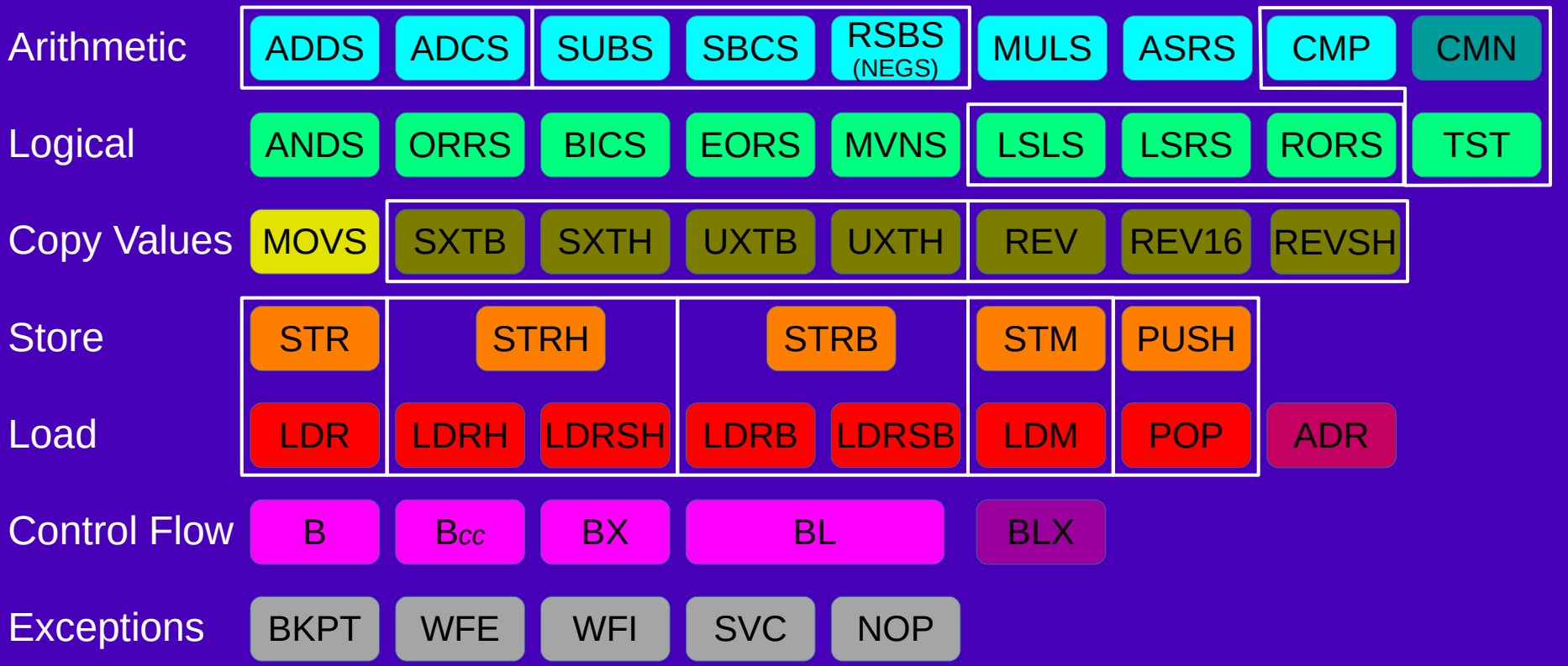


- There are many flavors of ARM Cortex-M CPUs.
  - We're using M0.
  - Not using M3.
    - Which is based on ARMv7.
  - Nor using M4.
  - Nor others.
- But unified assembly language covers all of them.
- And so does your book.

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.

Instruction Synopsis



Several instructions allow immediate values...

- 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)

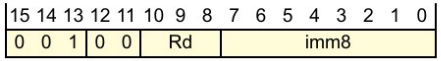
MOV (immediate) instruction

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>
```




```
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
```

- Looks almost like ADDS Rdn, #imm8.
  - The opcode is different.
  - Sets the flags, but neither Carry nor oVerflow.
- 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 for when using immediate operand.

Examples of Immediate Operands

"Example 0" on lecture notes web page

```
.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
```

- Note that we use an 'S' suffix with every instruction except CMP.
- The maximum size for the immediate value varies between instructions.

How to initialize a register with a 32-bit value

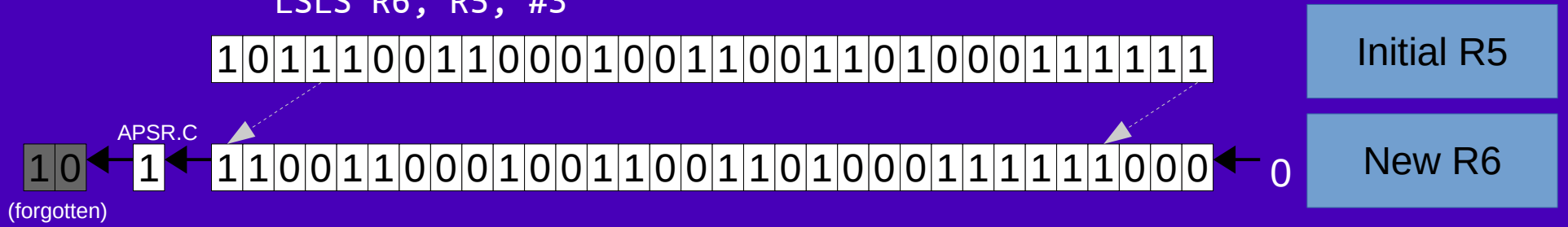
- You won't understand how this works at this point… You just need to use it.
- We can use an assembler trick to load any value into R0 – R7. For example:
  - LDR R0, =0x12345678
  - Note the "=" sign.

Logical Shift Left

- We can shift a 32-bit value in a register “left” by any number of bits.
  - This is the same as the C operator: <<
  - Bits that are shifted out the left end (MSB) of the register are moved into the APSR Carry flag, so we call it “LSLS”.
  - Zeros are shifted in on the right side (LSB).

Example:

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



What would this mean?

- LSLS R6,R5,#0

Register Addressing

- Easiest mode to understand.
  - "Use the value in the specified register."
  - But there are some interesting nuances.

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;
```

MOV R12, R11
010001 10 1 1011 100
0x46DC

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;
```

MOVS R6, R5
000 00 00000 101 110
0x002E

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.

Do we need MOVS Rd,Rm ?

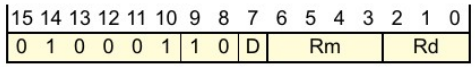
- Imagine you’re an engineer designing a CPU.
- You need an instruction to copy a value from one register to another. (and set the flags)
- You realize that you can use LSLS Rd,Rm,#0
- No need for a second instruction!
  - This will reduce the CPU cost...
    - Saving the company millions over time...
      - Which should, therefore, go into your paycheck...

A6.7.40 MOV (register)

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

Encoding T1 ARMv6-M, ARMv7-M, if <Rd> and <Rm> both from R0-R

MOV <Rd>,<Rm> Otherwise all versions of the Thumb instruction set.

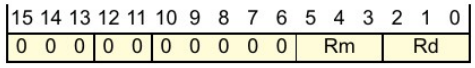


MOV R12, R11
010001 10 1 1011 100
0x46DC

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

Encoding T2 All versions of the Thumb instruction set.

MOVS <Rd>,<Rm>



MOVS R6, R5
000 00 00000 101 110
0x002E

```
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 n 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 <Rm> is the SP or PC.

Note

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

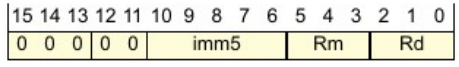
- ones in which <Rd> is the SP or PC and <Rm> is also the SP or PC
- ones in which S is specified and <Rm> is the SP, or <Rm> is the PC.

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>




```
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

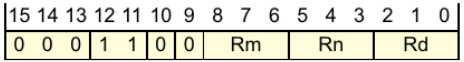
None.

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.

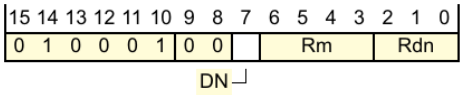
```
ADDS <Rd>,<Rn>,<Rm>
```




```
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>
```




```
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.

Operation

```
if ConditionPassed() then
    EncodingSpecificOperations();
    shifted = Shift(R[m], shift_t, shift_n, APSR.C);
    (result, carry, overflow) = AddWithCarry(R[n], shifted, '0');
    if d == 15 then
        ALUWritePC(result);  // setflags is always FALSE here
    else
        R[d] = result;
        if setflags then
            APSR.N = result<31>;
            APSR.Z = IsZeroBit(result);
            APSR.C = carry;
            APSR.V = overflow;
```

Exceptions

None.

- Instructions that can refer to all 16 registers:
  - ADD
  - CMP
  - MOV

Also an ADC instruction

- Add with Carry.
  - The standard ADD instruction ignores APSR.C as an input, but sets APSR.C to the carry-out value of the addition result.
  - ADC uses the APSR.C as a "carry-in"
  - We can use this for multi-word arithmetic.
- Written ADCS because it updates the flags.

Examples of Register Operands

```
.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
```

"Example 1" on lecture notes web page

- Note that we use an 'S' suffix with every instruction except TST.

Common Mistakes

- 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
    ```

Remember the reading assignment!

- Textbook, Chapter 4, “Arithmetic and Logic”, pages 75 – 96.
  - This will give you a much more thorough overview of the instructions for doing arithmetic, logic, shifting, etc.
  - Just watch out for all of those 32-bit instructions, which are available with the ARM Cortex-M4, that we don’t have on the Cortex-M0.