Instruction Sets
Immediate & Register Modes

ECE 362
https://engineering.purdue.edu/ee362/
Reading Assignment

  - We’re using an ARM Cortex-M0 (ARMv6) for this class. Not an ARMv7.

• Textbook, Chapter 4, “Arithmetic and Logic”, pages 75 – 96.
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] \implies \text{mov 0x40(%rdi,%rsi,4),%eax} \implies 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:
    • 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

• Cortex M0 CPUs use the "Thumb2" ISA.
• Only 56 different instructions.
  - 50 instructions are 16 bits long.
  - 6 instructions are 32 bits long.
    • Actually 16-bit instructions that tell the CPU there is one more 16-bit chunk.
• 1-, 2-, and 3-operand instructions.
• Standard operand order is right-to-left.
  - e.g. `adds r5,r2,r4` means `r5 = r2 + r4`
Addressing Modes

• The simple computer used what we call "absolute addressing"
  - The operand referred to a specific memory location.

• Many CPUs have a way to use absolute addressing. ARM Cortex M0 does not.
  - Each instruction is always 2 (or 4) bytes long
  - No room in the instruction for 32-bit address.
Cortex-M0 Instruction Set

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

• Four major addressing modes:
  – Immediate: small value contained in instruction
  – Register: src/dst register in instruction
  – Offset: value is at address relative to a register
  – Indexed: value is at address specified by register
A5.2 16-bit Thumb instruction encoding

The encoding of 16-bit Thumb instructions is:

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

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

• Small integer encoded directly into instruction.

  e.g.

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

  ADDS R3, #255  
  001 10 011 1111111  
  0x33FF

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?
- 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*

---

**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 R6, R2, #5

000 11 1 0 101 010 110

0x1D56

01 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

\[ d = \text{UInt}(Rd); \ n = \text{UInt}(Rn) ; \ \text{setflags} = !\text{InITBlock}() ; \ \text{imm32} = \text{ZeroExtend}(\text{imm3}, 32) ; \]

### Encoding T2

All versions of the Thumb instruction set.

ADDS R3, #255

001 10 011 11111111

0x33FF

01 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0 0 1 1 1 0 Rdn imm8

\[ d = \text{UInt}(Rdn); \ n = \text{UInt}(Rdn) ; \ \text{setflags} = !\text{InITBlock}() ; \ \text{imm32} = \text{ZeroExtend}(\text{imm8}, 32) ; \]

### 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>.

<r> 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.
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.
ADD S <Rd>, <Rn>, #<imm3>

| 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| 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.
ADD S <Rd>, #<imm8>

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

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

**Assembler syntax**

ADD{<op>}{<Rd>,} <Rn>, #<const> All encodings permitted

where:

S The instruction updates the flags.

<{op}> 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

EncodingsSpecificOperations();

(result, carry, overflow) = AddWithCarry(R[n], imm32, '0');
R[d] = result;
if setflags then

APSR.N = result<32>;
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 <> suffix must be omitted or AL in all instruction mnemonics except B<>
- in the pseudocode in this manual:
  - any reference to ITBlock() returns FALSE
  - any reference to LastITBlock() 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.
  - 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.
Several instructions allow immediate values...

- **ADD**: Add a number
- **ASR**: Arithmetic Shift Right by a number of bits
- **CMP**: Compare to a number
- **LSL**: Logical Shift Left by a number of bits
- **LSR**: Logical Shift Right by a number of bits
- **MOV**: Move immediate number into a register
- **ROR**: Rotate Right by a number of bits
- **RSB**: Reverse Subtract a number (result = \#imm – Rn) (only for \#0!)
- **SUB**: Subtract a number (result = Rn - \#imm)
MOV (immediate) instruction

• 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.
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: `LSLS R0, R0, #3`

```
Initial R0
1011110011000100110011011000111111

APSR.C
1
(forgotten)

New R0
110011000100110011010100011111110000
```
What would this mean?

- LSLS R0,R0,#0
Register Addressing

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

ARMv6-M, ARMv7-M, if $r8d$ and $r8m$ both from R0-R7. Otherwise all versions of the Thumb instruction set.

```
MOV R12, R11
010001 10 1 1011 100
0x46DC
```

```
MOV R6, R5
000 00 00000 101 110
0x002E
```

**Assemble 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.
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 condition flags based on the value.

**Encoding T1**

MOV \( \langle Rd \rangle, \langle Rm \rangle \)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>D</td>
<td>m</td>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( d = \text{UInt}(D; Rd); \ m = \text{UInt}(Rm); \ \text{setflags} = \text{FALSE}; \)

**Encoding T2**

All versions of the Thumb instruction set.

MOV \( \langle Rd \rangle, \langle Rm \rangle \)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Rm</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\( d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ \text{setflags} = \text{TRUE}; \)

**Assembler syntax**

MOV(S)\{<q>\} \( \langle Rd \rangle, \langle Rm \rangle \)

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.

\( \langle Rm \rangle \)  The source register. This register can be the SP or PC. The instruction must not specify \( S \) or \( Rm \) is the PC.

\( \langle Rd \rangle \)  The destination register. This register can be the SP or PC, provided \( S \) is not specified. If \( \langle Rd \rangle \) is the PC, the instruction causes a branch to the address moved to the PC.

---

**Note**

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

- ones in which \( \langle Rd \rangle \) is the SP or PC and \( \langle Rm \rangle \) is also the SP or PC
- ones in which \( S \) is specified and \( \langle Rm \rangle \) is the SP or \( S \) 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.

LSL \( \langle Rd \rangle, \langle Rm \rangle, \#<imm5> \)

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>imm5</td>
<td>Rd</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

if \( imm5 = '00000' \) then see MOV (register);

\( d = \text{UInt}(Rd); \ m = \text{UInt}(Rm); \ \text{setflags} = \text{setflags} \& \text{uint8Block}(); \)

if \( \text{setflags} \) then

\( \text{APSR.N} = \text{result}<31>; \)

\( \text{APSR.Z} = \text{IsZeroBit}(\text{result}); \)

\( \text{APSR.C} = \text{carry}; \)

// APSR.V unchanged

**Assembler syntax**

LSL\{<op>\} \( \langle Rd \rangle, \langle Rm \rangle, \#<imm5> \)

where:

\{<op>\}  The instruction updates the flags.

\{<op>\}  See *Standard assembler syntax fields* on page A6-98.

\( \langle Rd \rangle \)  The destination register.

\( \langle Rm \rangle \)  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();

\( \text{result, carry} = \text{Shift.C}(\text{R[m]}, \text{SRType.LSL, shift.n, APSR.C}); \)

\( R[d] = \text{result}; \)

if \( \text{setflags} \) then

\( \text{APSR.N} = \text{result}<31>; \)

\( \text{APSR.Z} = \text{IsZeroBit}(\text{result}); \)

\( \text{APSR.C} = \text{carry}; \)

// 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.

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 1 1 0 0</td>
</tr>
</tbody>
</table>
```

\[ d = \text{UInt}(\text{Rd}); \quad n = \text{UInt}(\text{Rn}); \quad m = \text{UInt}(\text{Rm}); \quad \text{setflags} = \text{!InITBlock}(); \]

\[ (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{\text{LSL}}, 0); \]

Encoding T2

All versions of the Thumb instruction set.

```
<table>
<thead>
<tr>
<th>15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 0 0 1 0 0</td>
</tr>
</tbody>
</table>
```

\[ \text{DN} = \text{J} \]

if (DN:Rdn) == '1101' || Rm == '1101' then SEE ADD (SP plus register);

\[ d = \text{UInt}(\text{DN}:\text{Rdn}); \quad n = d; \quad m = \text{UInt}(\text{Rm}); \quad \text{setflags} = \text{FALSE}; \quad (\text{shift}_t, \text{shift}_n) = (\text{SRType}_{\text{LSL}}, 0); \]

if n == 15 && m == 15 then UNPREDICTABLE;

if d == 15 && \text{InITBlock}() && \text{!LastInITBlock}() then UNPREDICTABLE;

Assembler syntax

```
ADD{} \{<Rd>,} \{<Rn>, <Rm>
```

where:

- \{<Rd>\}  
  If present, specifies that the instruction updates the flags. Otherwise, the instruction does not update the flags.

- \{<Rn>, <Rm>\}  
  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 \<Rm>\ 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 \<Rm>\ 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.

Exceptions

None.

Operation

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

• ADD: similar to MOV.
Also an ADC instruction

• Add with Carry.
  – The standard ADD instruction ignores the initial carry flag and sets it to the carry-out value.
  – ADC uses the C flag as a "carry-in"
  – We can use this for multi-word arithmetic.

• Written ADCS because it updates the flags.
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 available with the ARM Cortex-M3.