Demo program from Lecture #1

- Add four numbers together
- \( \text{total} = a + b + c + d \)
- \( \text{total}, a, b, c, \) and \( d \) are stored in memory
- operations (move and add) are performed in CPU
- how many memory ↔ CPU transfers?

Accessing memory is slow relative to the speed at which the processor can execute instructions

Processors use small fast internal storage to temporarily store values – called registers
ARM7TDMI Registers

- **ARM7TDMI Registers**
  - 15 word-size registers, labelled `r0`, `r1`, ..., `r14`
  - Program Counter Register, **PC**, also labelled **r15**
  - Current Program Status Register (**CPSR**)  

- Program Counter always contains the address in memory of the next instruction to be fetched

- CPSR contains information about the result of the last instruction executed (e.g. Was the result zero? Was the result negative?) and the status of the processor

- `r13` and `r14` are normally reserved for special purposes and you should avoid using them
A program is composed of a sequence of instructions stored in memory as **machine code**

- Instructions determine the operations performed by the processor (e.g. add, move, multiply, subtract, compare, ...)

A single instruction is composed of

- an **operator (instruction)**
- zero, one or more **operands**

E.g. ADD the values in r1 and r2 and store the result in r0

- Operator is ADD
- Want to store the result in r0 (first operand)
- We want to add the values in r1 and r2 (second and third operands)

Each instruction and its operands are encoded using a unique value

- E.g. 0xE0810002 is the machine that causes the processor to add the values in r1 and r2 and store the result in r3
Writing programs using **machine code** is possible but not practical.

Instead, we write programs using **assembly language**:
- Instructions are expressed using **mnemonics**
  - e.g. the word “ADD” instead of the machine code 0xE08
  - e.g. the expression “r2” to refer to register number two, instead of the machine code value 0x2

Assembly language must still be translated into machine code:
- Done using a program called an **assembler**
- Machine code produced by the assembler is stored in memory and executed by the processor.
Program 1.1 revisited

start

MOV r0, r1 ; Make the first number the subtotal
ADD r0, r0, r2 ; Add the second number to the subtotal
ADD r0, r0, r3 ; Add the third number to the subtotal
ADD r0, r0, r4 ; Add the fourth number to the subtotal

stop B stop
```
<table>
<thead>
<tr>
<th></th>
<th>Address</th>
<th>Machine Code</th>
<th>Original Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>00000000</td>
<td>AREA</td>
<td>Demo, CODE, READONLY</td>
</tr>
<tr>
<td>2</td>
<td>00000000</td>
<td>IMPORT</td>
<td>main</td>
</tr>
<tr>
<td>3</td>
<td>00000000</td>
<td>EXPORT</td>
<td>start</td>
</tr>
<tr>
<td>4</td>
<td>00000000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>00000000</td>
<td>start</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>00000000</td>
<td>E1A00001</td>
<td>MOV r0, r1</td>
</tr>
<tr>
<td>7</td>
<td>00000004</td>
<td>E0800002</td>
<td>ADD r0, r0, r2</td>
</tr>
<tr>
<td>8</td>
<td>00000008</td>
<td>E0800003</td>
<td>ADD r0, r0, r3</td>
</tr>
<tr>
<td>9</td>
<td>0000000C</td>
<td>E0800004</td>
<td>ADD r0, r0, r4</td>
</tr>
<tr>
<td>10</td>
<td>00000010</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>00000010</td>
<td>EAFFFFFFE</td>
<td>stop B stop</td>
</tr>
<tr>
<td>12</td>
<td>00000014</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>00000014</td>
<td>END</td>
<td></td>
</tr>
</tbody>
</table>
```
Every ARM machine code instruction is 32-bits long

32-bit instruction word must encode
- operation (instruction)
- all the required instruction operands

Example – add r0, r0, r2

```
1 1 1 0 0 0 0 0 1 0 0 0
31 20 19 16 15 12 11 4 3 0
```

```
1 1 1 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0
```

```
E 0 8 0 0 0 0 0 0 2
31 28 27 24 23 20 19 16 15 12 11 8 7 4 3 0
```
Machine Code and Assembly Language

3. Execute the instruction

1. Fetch instruction at PC address

2. Decode the instruction

32 bits = 4 bytes = 1 word

Memory

```
0xA0000134
0xA0000138
0xA000013C
0xA0000140
0xA0000144
0xA0000148
0xA000014C
0xA0000150
0xA0000154
0xA0000158
0xA000015C
0xA0000160
0xA0000164
0xA0000168
0xA000016C
0xA0000170
0xA0000174
```

1. Fetch instruction at PC address

2. Decode the instruction

3. Execute the instruction

ADD R0, R0, R3
Program Execution

3D1 / Microprocessor Systems I
ARM Assembly Language

Start Debug Session

Program assembled and loaded into memory at a pre-defined address

Program Counter (PC) set to same pre-defined address

Fetch-Decode-Execute cycle resumes

What happens when we reach the end of our program?

Memory

32 bits = 4 bytes = 1 word

PC

0xFFFFF8F
0xFFFFF8C
0xA0000004
0xA0000008
0xA000000C
0xA0000010
0xA0000014

...
Program 3.1 – Swap Registers

- Write an assembly language program to swap the contents of register r0 and r1

```
start
  MOV  r2, r0 ; temp <-- r0
  MOV  r0, r1 ; r0 <-- r1
  MOV  r1, r2 ; r1 <-- temp
stop  B  stop
```

```
start
  EOR  r0, r0, r1 ; r0 <-- r0 xor r1
  EOR  r1, r0, r1 ; r1 <-- (r0 xor r1) xor r1 = r0
  EOR  r0, r0, r1 ; r0 <-- (r0 xor r1) xor r0 = r1
stop  B  stop
```

Compare both programs with respect to instructions executed and registers used ...
Immediate Operands

- Register operands

  \[
  \text{ADD } Rd, \ Rn, \ Rm \\
  \text{MOV } Rd, \ Rm
  \]

- Often want to use constant values as operands, instead of registers

  \[
  \text{ADD } Rd, \ Rn, \ #x \\
  \text{MOV } Rd, \ #x
  \]

  - e.g. Move the value 0 (zero) into register r0

    \[
    \text{MOV } r0, \ #0 \quad ; \ r0 \leftarrow 0
    \]

  - e.g. Set r1 = r2 + 1

    \[
    \text{ADD } r1, \ r2, \ #1 \quad ; \ r1 \leftarrow r2 + 1
    \]

- Called an immediate operand, syntax \#x
Write an assembly language program to compute ...

\[ 4x^2 + 3x \]

... if \( x \) is stored in \( r1 \). Store the result in \( r0 \)

```
start
    MUL r0, r1, r1 ; result <-- x * x
    LDR r2, =4 ; tmp <-- 4
    MUL r0, r2, r0 ; result <-- 4 * x * x
    LDR r2, =3 ; tmp <-- 3
    MUL r2, r1, r2 ; tmp <-- x * tmp
    ADD r0, r0, r2 ; result <-- result + tmp
stop
    B stop
```

• Cannot use `MUL` to multiply by a constant value
• `MUL Rx, Rx, Ry` produces unpredictable results [UPDATE]
• \( r1 \) unmodified ... which may be something we want ... or not
Note use of operand =3

- Move constant value 3 into register r2

Load Register instruction can be used to load any 32-bit signed constant value into a register

Note use of =x syntax instead of #x with LDR instruction
## MOV, LDR and Constant Values

**ARM Assembly Language**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV</td>
<td>r0</td>
<td>#7</td>
</tr>
<tr>
<td>LDR</td>
<td>r0</td>
<td>=7</td>
</tr>
<tr>
<td>MOV</td>
<td>r0</td>
<td>#0x4FE8</td>
</tr>
<tr>
<td>LDR</td>
<td>r0</td>
<td>=0x4FE8</td>
</tr>
</tbody>
</table>

### Examples

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>MOV</td>
<td>r0, #7</td>
</tr>
<tr>
<td>00000000</td>
<td>LDR</td>
<td>r0, =7</td>
</tr>
<tr>
<td>00000000</td>
<td>MOV</td>
<td>r0, #0x4FE8</td>
</tr>
<tr>
<td>00000000</td>
<td>LDR</td>
<td>r0, =0x4FE8</td>
</tr>
</tbody>
</table>

### Notes
- Cannot fit large constant values in a 32-bit instruction
- LDR is a “pseudo-instruction” that simplifies the implementation of a work-around for this limitation
- For small constants, LDR is replaced with MOV
Assembly Language Programming Guidelines

- Provide **meaningful** comments and assume someone else will be reading your code

```
MUL    r2, r1, r2 ; r2 <-- r1 * r2
```

```
MUL    r2, r1, r2 ; tmp <-- x * tmp
```

- Break your programs into small pieces
- While starting out, keep programs simple
- Pay attention to initial values in registers (and memory)