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
ARMv5te Registers

- **armv5te Registers**
  - 15 word-size registers, labelled \( r_0, r_1, ..., r_{14} \)
  - Program Counter Register, \( PC \), also labelled \( r_{15} \)
  - 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
- \( r_{13} \) and \( r_{14} \) 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

ARM Assembly Language

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
Program 1.1 – Demonstration (Demo.lst)

4 00000000
5 00000000
6 00000000 E1A00001 MOV r0, r1
7 00000004 E0800002 ADD r0, r0, r2
8 00000008 E0800003 ADD r0, r0, r3
9 0000000C E0800004 ADD r0, r0, r4
10 00000010
11 00000010 EAFFFFFE stop: B stop
12 00000014
13 00000014 END

Big Endian
Little Endian

R00=60000000  R01=00000000  R02=00000000  R03=00000000  >  3 0000 0100A0E1
R04=60000000  R05=00000000  R06=00000000  R07=00000000  4 0004 020080E0
R08=60000000  R09=00000000  R10=00000000  R11=00000000  5 0008 030080E0
R12=60000000  R13=00000000  R14=00000000  R15=00000000  6 000c 040080E0
PSR=4000001d3 -Z--  8 0010 FEFFFFEA  stop: B stop

0xA1600400: 00 00 00 00 00 00 00 00
0xA1600408: 00 00 00 00 00 00 00 00
0xA1600410: 00 00 00 00 00 00 00 00
0xA1600418: 00 00 00 00 00 00 00 00
0xA1600420: 00 00 00 00 00 00 00 00
0xA1600428: 00 00 00 00 00 00 00 00
0xA1600430: 00 00 00 00 00 00 00 00
0xA1600438: 00 00 00 00 00 00 00 00
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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Rn</th>
<th>Rd</th>
<th>Rm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000000001000</td>
<td>20</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td>0000000001000</td>
<td>0000</td>
<td>0000</td>
<td>000000000</td>
</tr>
</tbody>
</table>

E 0 8 0 0 0 0 2
Machine Code and Assembly Language

1. Fetch instruction at PC address
2. Decode the instruction
3. Execute the instruction

32 bits = 4 bytes = 1 word

Memory

ADD R0, R0, R3
Write an assembly language program to swap the contents of register \( r0 \) and \( r1 \)

**Program 3.1 – Swap Registers**

- **ARM Assembly Language**

```
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
  - ADD Rd, Rn, Rm
  - MOV Rd, Rm

- Often want to use constant values as operands, instead of registers
  - ADD Rd, Rn, #x
  - MOV Rd, #x

  - e.g. Move the value 0 (zero) into register r0
    
    ```
    MOV r0, #0 @ r0 <-- 0
    ```

  - e.g. Set r1 = r2 + 1
    
    ```
    ADD r1, r2, #1 @ r1 <-- 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 \( r0 \)

- Cannot use MUL to multiply by a constant value
- MUL \( Rx, Rx, Ry \) produces unpredictable results
### Load Register

#### ARM Assembly Language

```
LDR   r2, =3   @ tmp <-- 3
MUL   r2, r1, r2  @ tmp <-- x * tmp
```

- **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

```
LDR   r4, =0x000013C @ r4 <-- 0x000013C
```

- **Note use of** `=x` **syntax instead of** `#x` **with LDR instruction**
- 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

  \[
  \text{MUL } r2, r1, r2 \quad \text{@ } r2 \leftarrow r1 \times r2
  \]

  \[
  \text{MUL } r2, r1, r2 \quad \text{@ } \text{tmp} \leftarrow x \times \text{tmp}
  \]

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