Compiler Development (CMPSC 401)

ARM Architecture

Janyl Jumadinova

April 4, 2019
ARM Ltd

- Founded in November 1990
  - Spun out of Acorn Computers
- Designs the ARM range of RISC processor cores.
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers.
  - ARM does not fabricate silicon itself.
ARM Ltd

- Founded in November 1990
  - Spun out of Acorn Computers
- Designs the ARM range of RISC processor cores.
- Licenses ARM core designs to semiconductor partners who fabricate and sell to their customers.
  - ARM does not fabricate silicon itself.
Also develop technologies to assist with the designing of the ARM architecture
- Software tools, boards, debug hardware, application software, graphics, bus architectures, peripherals, cell libraries
The Architecture for the Digital World

ARM designs technology that lies at the heart of advanced digital products
ARM Business Today

- Processor Shipped In Total: > 50 Billion
- Processor Licenses: 500+
- Semiconductor Partners: 200+
- Process Technology: 16 – 250 nm
- Connected Community Members: 700+
ARM Processor Applications
Phoenix

Wireless Sensor Network

- Sensors, timers
- Cortex-M0 + 16KB RAM 65nm UWB Radio antenna
- 10 kB Storage memory ~3fW/bit
- 12μAh Li-ion Battery

Wirelessly networked into large scale sensor arrays

Battery

Solar Cells

Processor, SRAM and PMU

Also, Kinetis KL02 (chip size: 1.9 x 2.0 millimeters)
World’s Largest ARM Computer?

IceCube

4200 ARM powered Neutrino Detectors

70 bore holes 2.5km deep
60 detectors per string starting 1.5km down 2.5km
1km³ of active telescope

Work supported by the National Science Foundation and University of Wisconsin-Madison
From $1mm^3$ to $1km^3$

The Architecture for the Digital World
The ARM is a 32-bit architecture.

When used in relation to the ARM:
  - Byte means 8 bits
  - Halfword means 16 bits (two bytes)
  - Word means 32 bits (four bytes)

Most ARM’s implement two instruction sets:
- 32-bit ARM Instruction Set
- 16-bit/32bit Thumb Instruction Set
The Registers

- ARM has 37 registers in total
- All are 32-bit long
  - 1 dedicated program counter (r15)
  - 1 dedicated current program status register
  - 5 dedicated saved program status registers
  - 30 general purpose registers
## Register Allocation

### General registers and Program Counter

<table>
<thead>
<tr>
<th>User32 / System</th>
<th>FIQ32</th>
<th>Supervisor32</th>
<th>Abort32</th>
<th>IRQ32</th>
<th>Undefined32</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
<td>r0</td>
</tr>
<tr>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
<td>r1</td>
</tr>
<tr>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
<td>r2</td>
</tr>
<tr>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
<td>r3</td>
</tr>
<tr>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
<td>r4</td>
</tr>
<tr>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
<td>r5</td>
</tr>
<tr>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
<td>r6</td>
</tr>
<tr>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
<td>r7</td>
</tr>
<tr>
<td>r8</td>
<td>r8_fiq</td>
<td>r8</td>
<td>r8</td>
<td>r8</td>
<td>r8</td>
</tr>
<tr>
<td>r9</td>
<td>r9_fiq</td>
<td>r9</td>
<td>r9</td>
<td>r9</td>
<td>r9</td>
</tr>
<tr>
<td>r10</td>
<td>r10_fiq</td>
<td>r10</td>
<td>r10</td>
<td>r10</td>
<td>r10</td>
</tr>
<tr>
<td>r11</td>
<td>r11_fiq</td>
<td>r11</td>
<td>r11</td>
<td>r11</td>
<td>r11</td>
</tr>
<tr>
<td>r12</td>
<td>r12_fiq</td>
<td>r12</td>
<td>r12</td>
<td>r12</td>
<td>r12</td>
</tr>
<tr>
<td>r13 (sp)</td>
<td>r13_fiq</td>
<td>r13_svc</td>
<td>r13_abt</td>
<td>r13_irq</td>
<td>r13.Undef</td>
</tr>
<tr>
<td>r14 (lr)</td>
<td>r14_fiq</td>
<td>r14_svc</td>
<td>r14_abt</td>
<td>r14_irq</td>
<td>r14.Undef</td>
</tr>
<tr>
<td>r15 (pc)</td>
<td>r15 (pc)</td>
<td>r15 (pc)</td>
<td>r15 (pc)</td>
<td>r15 (pc)</td>
<td>r15 (pc)</td>
</tr>
</tbody>
</table>

### Program Status Registers

- cpsr
- spsr_fiq
- spsr_svc
- spsr_abt
- spsr_irq
- spsr.Undef
Program Status Registers (CPSR and SPSR)

![Program Status Register Diagram]

Copies of the ALU status flags (latched if the instruction has the "S" bit set).

* **Condition Code Flags**
  
  - N = Negative result from ALU flag.
  - Z = Zero result from ALU flag.
  - C = ALU operation Carried out
  - V = ALU operation Over/flowed

* **Mode Bits**
  
  M[4:0] define the processor mode.

* **Interrupt Disable bits.**
  
  - I = 1, disables the IRQ.
  - F = 1, disables the FIQ.

* **T Bit** *(Architecture v4T only)*
  
  - T = 0, Processor in ARM state
  - T = 1, Processor in Thumb state
## Condition Flags

<table>
<thead>
<tr>
<th>Flag</th>
<th>Logical Instruction</th>
<th>Arithmetic Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Negative (N=‘1’)</td>
<td>No meaning</td>
<td>Bit 31 of the result has been set</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indicates a negative number in signed operations</td>
</tr>
<tr>
<td>Zero (Z=‘1’)</td>
<td>Result is all zeroes</td>
<td>Result of operation was zero</td>
</tr>
<tr>
<td>Carry (C=‘1’)</td>
<td>After Shift operation ‘1’ was left in carry flag</td>
<td>Result was greater than 32 bits</td>
</tr>
<tr>
<td>oVerflow (V=‘1’)</td>
<td>No meaning</td>
<td>Result was greater than 31 bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Indicates a possible corruption of the sign bit in signed numbers</td>
</tr>
</tbody>
</table>
Conditional Execution

• Most instruction sets only allow branches to be executed conditionally.
• However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
Conditional Execution

- Most instruction sets only allow branches to be executed conditionally.
- However by reusing the condition evaluation hardware, ARM effectively increases number of instructions.
- All instructions contain a **condition** field which determines whether the CPU will execute them.
- Allows very dense in-line code, without branches.
- The time penalty of not executing several conditional instructions is frequently less than overhead of the branch or subroutine call that would otherwise be needed.
## Condition Flags

<table>
<thead>
<tr>
<th>Condition</th>
<th>0</th>
<th>0</th>
<th>I</th>
<th>OPCODE</th>
<th>S</th>
<th>Rn</th>
<th>Rs</th>
<th>OPERAND-2</th>
<th>Instruction Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EQ - Z set (equal)</td>
</tr>
<tr>
<td>0001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NE - Z clear (not equal)</td>
</tr>
<tr>
<td>0010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HS / CS - C set (unsigned higher or same)</td>
</tr>
<tr>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LO / CC - C clear (unsigned lower)</td>
</tr>
<tr>
<td>0100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>MI - N set (negative)</td>
</tr>
<tr>
<td>0101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>PL - N clear (positive or zero)</td>
</tr>
<tr>
<td>0110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VS - V set (overflow)</td>
</tr>
<tr>
<td>0111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>VC - V clear (no overflow)</td>
</tr>
<tr>
<td>1000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>HI - C set and Z clear (unsigned higher)</td>
</tr>
<tr>
<td>1001</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LS - C clear or Z (set unsigned lower or same)</td>
</tr>
<tr>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GE - N set and V set, or N clear and V clear (&gt; or =)</td>
</tr>
<tr>
<td>1011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LT - N set and V clear, or N clear and V set (&gt;)</td>
</tr>
<tr>
<td>1100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>GT - Z clear, and either N set and V set, or N clear and V set (&gt; =)</td>
</tr>
<tr>
<td>1101</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>LE - Z set, or N set and V clear, or N clear and V set (&lt;, or =)</td>
</tr>
<tr>
<td>1110</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>AL - always</td>
</tr>
<tr>
<td>1111</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>NV - reserved.</td>
</tr>
</tbody>
</table>
# Condition Flags

<table>
<thead>
<tr>
<th>Affix</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>eq</td>
<td>Z=1</td>
<td>Zero (EQual to 0)</td>
</tr>
<tr>
<td>ne</td>
<td>Z=0</td>
<td>Not zero (Not Equal to 0)</td>
</tr>
<tr>
<td>cs / hs</td>
<td>C=1</td>
<td>Carry Set / unsigned Higher or Same</td>
</tr>
<tr>
<td>cc / lo</td>
<td>C=0</td>
<td>Carry Clear / unsigned LOwer</td>
</tr>
<tr>
<td>mi</td>
<td>N=1</td>
<td>Negative (MInus)</td>
</tr>
<tr>
<td>pl</td>
<td>N=0</td>
<td>Positive or zero (PLus)</td>
</tr>
<tr>
<td>vs</td>
<td>V=1</td>
<td>Sign overflow (oVerflow Set)</td>
</tr>
<tr>
<td>vc</td>
<td>V=0</td>
<td>No sign overflow (oVerflow Clear)</td>
</tr>
<tr>
<td>hi</td>
<td>C=1 &amp; Z=0</td>
<td>Unsigned HLHiger</td>
</tr>
<tr>
<td>ls</td>
<td>C=0</td>
<td>Z=1</td>
</tr>
<tr>
<td>ge</td>
<td>N=V</td>
<td>Signed Greater or Equal</td>
</tr>
<tr>
<td>lt</td>
<td>N != V</td>
<td>Signed Less Than</td>
</tr>
<tr>
<td>gt</td>
<td>Z=0 &amp; N=V</td>
<td>Signed Greater Than</td>
</tr>
<tr>
<td>le</td>
<td>Z=1</td>
<td>N != V</td>
</tr>
<tr>
<td>al</td>
<td>-</td>
<td>ALways (default)</td>
</tr>
<tr>
<td>nv</td>
<td>-</td>
<td>NeVer</td>
</tr>
</tbody>
</table>
Condition Flags

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
- For example an add instruction takes the form:
  
  \[
  \text{ADD } r0,r1,r2 \ ; \ r0 = r1 + r2 \ (\text{ADDAL})
  \]
Condition Flags

- To execute an instruction conditionally, simply postfix it with the appropriate condition:
- For example an add instruction takes the form:
  \[ \text{ADD } r0, r1, r2 \ ; \ r0 = r1 + r2 \ (\text{ADDAL}) \]
- To execute this only if the zero flag is set:
  \[ \text{ADDEQ } r0, r1, r2 \ ; \ If \ zero \ flag \ set \ then... \]
  \[ \quad ; \ ...r0 = r1 + r2 \]
Condition Flags

- By default, data processing operations do not affect the condition flags (apart from the comparisons where this is the only effect).
- To cause the condition flags to be updated, the S bit of the instruction needs to be set by postfixing the instruction (and any condition code) with an “S”.
- For example to add two numbers and set the condition flags:
  
  ```
  ADDS r0, r1, r2 ; r0 = r1 + r2
  ;... and set flags
  ```
Instruction Classes

- Branch instructions
- Data processing instructions
- Load and store instructions
- Status register access instructions
- Miscellaneous instructions
Branch Instructions

- **B** – Branch
  - Absolute branch to a target address, relative to Program Counter (PC)
  - +/- 256 bytes range, conditional execution supported
  - +/- 1MB range, no conditional execution supported

- **BL** – Branch with Link
  - Branch to a subroutine – Link register is updated
  - +/- 16MB range, relative to Program Counter (PC)
Branch Instructions

Branch : B{<cond>} label
Branch with Link : BL{<cond>} sub_routine_label
## Conditional Branches

<table>
<thead>
<tr>
<th>Branch</th>
<th>Interpretation</th>
<th>Normal uses</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>Unconditional</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BAL</td>
<td>Always</td>
<td>Always take this branch</td>
</tr>
<tr>
<td>BEQ</td>
<td>Equal</td>
<td>Comparison equal or zero result</td>
</tr>
<tr>
<td>BNE</td>
<td>Not equal</td>
<td>Comparison not equal or non-zero result</td>
</tr>
<tr>
<td>BPL</td>
<td>Plus</td>
<td>Result positive or zero</td>
</tr>
<tr>
<td>BMI</td>
<td>Minus</td>
<td>Result minus or negative</td>
</tr>
<tr>
<td>BCC</td>
<td>Carry clear</td>
<td>Arithmetic operation did not give carry-out</td>
</tr>
<tr>
<td>BLO</td>
<td>Lower</td>
<td>Unsigned comparison gave lower</td>
</tr>
<tr>
<td>BCS</td>
<td>Carry set</td>
<td>Arithmetic operation gave carry-out</td>
</tr>
<tr>
<td>BHS</td>
<td>Higher or same</td>
<td>Unsigned comparison gave higher or same</td>
</tr>
<tr>
<td>BVC</td>
<td>Overflow clear</td>
<td>Signed integer operation; no overflow occurred</td>
</tr>
<tr>
<td>BVS</td>
<td>Overflow set</td>
<td>Signed integer operation; overflow occurred</td>
</tr>
<tr>
<td>BGT</td>
<td>Greater than</td>
<td>Signed integer comparison gave greater than</td>
</tr>
<tr>
<td>BGE</td>
<td>Greater or equal</td>
<td>Signed integer comparison gave greater or equal</td>
</tr>
<tr>
<td>BLT</td>
<td>Less than</td>
<td>Signed integer comparison gave less than</td>
</tr>
<tr>
<td>BLE</td>
<td>Less or equal</td>
<td>Signed integer comparison gave less than or equal</td>
</tr>
<tr>
<td>BHI</td>
<td>Higher</td>
<td>Unsigned comparison gave higher</td>
</tr>
<tr>
<td>BLS</td>
<td>Lower or same</td>
<td>Unsigned comparison gave lower or same</td>
</tr>
</tbody>
</table>
Data Processing Instructions

- Largest family of ARM instructions, all sharing the same instruction format.
- Contains:
  - Arithmetic operations
  - Comparisons (no results just set condition codes)
  - Logical operations
  - Data movement between registers
Data Processing Instructions

- Remember, this is a load / store architecture
- These instruction only work on registers, NOT memory.
- They each perform a specific operation on one or two operands.
- First operand always a register \( Rn \)
- Second operand sent to the ALU via barrel shifter.
Data processing instructions

- Standard Data Processing Instructions
  - ADD, ADC, SUB, SBC, RSB
  - AND, ORR, EOR, BIC
  - MOV, MVN
  - TST, CMP, CMN
  - ADR (Pseudo Instruction)

- Shift and Rotate Instructions
  - ASR
  - LSL, LSR
  - ROR

- Multiply Instruction
  - MUL

- Sign/Unsign Extend Instructions
  - SXTB, SXTH, UXTB, UXTH

- Miscellaneous Data Processing
  - REV, REV16, REVSH

Examples:

- SUBS r0,#1  
  (r0 ← r0 - 1)

- ORRS r0,r1  
  (r0 ← r0 | r1)

- MOVs r0,#1  
  (r0 ← r0 + 1)

- CMP r0,r1

- RSBS r0,r1,#0  
  (r0 ← -r1)

- ADR r0, Start  
  (r0 ← [Start])

- ASRS r0,r1,#7  
  (r0 ← r1 >> 7)

- LSLS r0,r1,#3  
  (r0 ← r1 << 3)

- RORS r0,r1  
  (r0 ← r0 >> r1)

- MULS r0,r1,r0  
  (r0 ← r1 * r0)

- UXTB r0,r1  
  (r0 ← r1[7:0])

- REV r0,r1  
  Byte Swap
Arithmetic Operations

Operations are:
- **ADD**  operand1 + operand2  ; Add
- **ADC**  operand1 + operand2 + carry ; Add with carry
- **SUB**  operand1 - operand2  ; Subtract
- **SBC**  operand1 - operand2 + carry -1 ; Subtract with carry
- **RSB**  operand2 - operand1  ; Reverse subtract
- **RSC**  operand2 - operand1 + carry -1 ; Reverse subtract with carry

Syntax:
- `<Operation>{<cond>}{S} Rd, Rn, Operand2`

Examples
- **ADD** r0, r1, r2
- **SUBGT** r3, r3, #1
- **RSBLES** r4, r5, #5
Comparisons

The only effect of the comparisons is to update the condition flags.

**Operations are:**
- CMP  \( \text{operand}_1 - \text{operand}_2 \)  ; Compare
- CMN  \( \text{operand}_1 + \text{operand}_2 \)  ; Compare negative
- TST  \( \text{operand}_1 \text{ AND operand}_2 \)  ; Test
- TEQ  \( \text{operand}_1 \text{ EOR operand}_2 \)  ; Test equivalence

**Syntax:**
- \(<\text{Operation}>\{<\text{cond}>\} \text{ Rn, Operand}_2\)

**Examples:**
- CMP \( \text{r0, r1} \)
- TSTEQ \( \text{r2, #5} \)
Logical Operations

Operations are:
- `AND` operand1 AND operand2
- `EOR` operand1 EOR operand2
- `ORR` operand1 OR operand2
- `ORN` operand1 NOR operand2
- `BIC` operand1 AND NOT operand2 [ie bit clear]

Syntax:
- `<Operation>{<cond>}{S} Rd, Rn, Operand2`

Examples:
- `AND r0, r1, r2`
- `BICEQ r2, r3, #7`
- `EORS r1,r3,r0`
Example

C:
    if (i == 0)
    {
        i = i + 10;
    }

ARM: (assume i in R1)
    SUBS    R1, R1, #0
    ADDIEQ  R1, R1, #10
Example

C:
for ( i = 0 ; i < 15 ; i++)
{
    j = j + j;
}

ARM:
SUB R0, R0, R0 ; i -> R0 and i = 0
start CMP R0, #15 ; is i < 15?
ADDLT R1, R1, R1 ; j = j + j
ADDLT R0, R0, #1 ; i++
BLT start
Data Movement

Operations are:
MOV  operand2
MVN  NOT operand2

Note that these make no use of operand1.

Syntax:
- <Operation>{<cond>}{S} Rd, Operand2

Examples:
MOV  r0, r1
MOV$  r2, #10
MVNEQ r1,#0
Barrel Shifter

- The ARM doesn’t have actual shift instructions.
- Instead it has a **barrel shifter** which provides a mechanism to carry out shifts as part of other instructions.
- So what operations does the barrel shifter support?
Barrel Shifter - Left Shift

Shifts left by the specified amount (multiplies by powers of two)

\[ \text{LSL} \#5 \implies \text{multiply by 32} \]
Barrel Shifter - Right Shifts

**Logical Shift Right (LSR)**
Shifts right by the specified amount (divides by powers of two) e.g.

\[ \text{LSR } \#5 = \text{divide by 32} \]

**Arithmetic Shift Right (ASR)**
Shifts right (divides by powers of two) and preserves the sign bit, for 2's complement operations. e.g.

\[ \text{ASR } \#5 = \text{divide by 32} \]
Barrel Shifter - Rotations

**Rotate Right (ROR)**

Similar to an ASR but the bits wrap around as they leave the LSB and appear as the MSB.

e.g. ROR #5

Note the last bit rotated is also used as the Carry Out.

**Rotate Right Extended (RRX)**

This operation uses the CPSR C flag as a 33rd bit.

Rotates right by 1 bit. Encoded as ROR #0
Load and Store instructions

- **Unsigned Loads/Stores**
  - LDR/STR
  - LDRH/STRH
  - LDRB/STRB

- **Signed Loads**
  - LDRSH
  - LDRSB

- **Load/Stores Multiple**
  - LDM, LDMIA/LDMFD (same as LDM, but with base register update option)
  - STM, STMIA/STMEA (same as STM, but with base register update option)
  - PUSH, POP
    - Uses "Full Descending (FD)" stack, SP always points to last filled data and SP predecrements for each PUSH)

- **Examples**
  - LDR r0, [r1] (r0 ← [r1])
  - STM r0, {r1, r2} (r1 → [r0]) (r2 → [r0+4])
  - LDM r0, {r1, r2} (r1 ← [r0]) (r2 ← [r0+4])
  - PUSH {r1, r2} (r1 → [SP], r2 → [SP+4])
  - POP {r1, r2} (r1 ← [SP], r2 ← [SP-4])
C:
    z = (a << 2) | (b & 15);

ARM:
ADR r4,a               ; get address for a
LDR r0,[r4]             ; get value of a
MOV r0,r0,LSL#2         ; perform shift
ADR r4,b               ; get address for b
LDR r1,[r4]             ; get value of b
AND r1,r1,#15          ; perform AND
ORR r1,r0,r1           ; perform OR
ADR r4,z               ; get address for z
STR r1,[r4]             ; store value for z
Running ARM code

We will test ARM programs on a bare metal emulated by Qemu. The assembly program source file consists of a sequence of statements, one per line. Each statement has the following format.

label: instruction @ comment
Running ARM code: add.s

```
.text

start: @ Label
    mov  r0, #5 @ Load register r0 with value 5
    mov  r1, #4 @ Load register r1 with value 4
    add  r2, r1, r0 @ Add r0 and r1, store in r2

stop:   b stop @ Infinite loop to stop
e        execution
```

The .text is an assembler directive, which says that the following instructions have to be assembled into the code section.
Running ARM code: add.s

To assemble the program, we can invoke the GNU Toolchain’s assembler as:

```
arm-none-eabi-as -o add.o add.s
```
Running ARM code: add.s

To assemble the program, we can invoke the GNU Toolchain’s assembler as:

```
arm-none-eabi-as -o add.o add.s
```

To generate the executable file, we can invoke the GNU Toolchain’s linker ld:

```
arm-none-eabi-ld -Ttext=0x0 -o add.elf add.o
```

`-Ttext=0x0`, specifies that addresses should be assigned to the labels, such that the instructions were starting from address 0x0. To view the address assignment for various labels, use `nm` command: `arm-none-eabi-nm add.elf`
Running ARM code

The ELF format works when you have an OS around, but to run the program on bare metal, we will have to convert it to a binary format.
- Use GNU toolchain’s `objcopy` command can be used to convert between different object file formats:

```
objcopy -O <output-format> <in-file> <out-file>

arm-none-eabi-objcopy -O binary add.elf add.bin
```
Running ARM code

- We will run our program on the connex board.
- On this board a 16MB Flash is located at address 0x0.
- When qemu emulates the connex board, a file has to be specified which will be treated file as Flash memory.
- To test the program, on the emulated Gumstix connex board, we first create a 16MB file representing the Flash.
- We use the dd command to copy 16MB of zeroes from /dev/zero to the file flash.bin. The data is copied in 4K blocks.

```bash
dd if=/dev/zero of=flash.bin bs=4096 count=4096
```
Running ARM code

- We now copy `add.bin` file into the beginning of the Flash:
  `dd if=add.bin of=flash.bin bs=4096 conv=notrunc`
- This is the equivalent of programming the bin file on to the Flash memory.
- After reset, the processor will start executing from address 0x0, and the instructions from the program will get executed. The command to invoke qemu is:
  `qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null`
Running ARM code

To view the contents of the registers the info registers monitor command can be used.

(qemu) info registers
R00=00000005  R01=00000004  R02=00000009  R03=00000000
R04=00000000  R05=00000000  R06=00000000  R07=00000000
R08=00000000  R09=00000000  R10=00000000  R11=00000000
R12=00000000  R13=00000000  R14=00000000  R15=0000000c
PSR=400001d3  -Z--  A svc32