Lecture 8 ARM Instruction Set Architecture

- In this lecture, we will consider some aspects of the ARM instruction set architecture (ISA) in detail.
- We shall consider the format of some instruction codes and their relationship with the assembly instruction.
- You will understand the role of the assembler.
- We start with a simple Branch or Branch with Link instruction:
  - B{condition}  <address>
  - BL{condition} <address>
- Bits [27:25] identify this as a B or BL instruction – they have values 101 only for these instructions.

```
31  28  27  25  24  23  0
cond 1 0 1  L  24-bit signed word offset
```

Branch and Branch with Link

- The top 4 bits [31:28] are used to specify the conditions under which the instruction is executed – this is common with all other instructions.
- The L-bit (bit 24) is set if it is a branch with link instruction and clear if it is a plain branch.
  - BL is jump to subroutine instruction - r14 <- return address.
- 24-bit signed offset specifies destination of branch in 2's complement form. The word offset is shifted left by 2 bits to form a byte offset.
  - This offset is added to the PC by the processor.
- The range of a branch is approx +/- 32 Mbytes: \(2^{23} \times 4 = 32\)M.
### B and BL: Examples

**BEQ label1**

... 

**BL label2**

**label1**

... 

**label2**

... 

- Let’s assume that label1 and label2 correspond to addresses 0x1C30 and 0x2C30, respectively.
- Further, assume that the address of the “BEQ” instruction is 0x0100 and of the “BL” instruction is 0x0120.
- Because of the pipelining on the ARM, when the “BEQ” instruction is *executing*, the instruction being *fetched* will be two instructions later.
- Two instructions = 2*4 = 8 bytes
- So byte offset for BEQ = 0x1C30 – 0x0100 – 8 = 0x1B28
B and BL: examples

- So word offset for BEQ instruction is \(0x1B28 / 4 = 0x6CA\)
- Byte offset for BL instruction is \(0x23C0 - 0x100 - 8 = 0x22B8\)
- So word offset for BL instruction is \(0x22B8 / 4 = 0x8AE\)

- Machine code representations:
  - \(\text{BEQ label1} => 0000\ 101\ 0\ 0000\ 0000\ 0000\ 0110\ 1100\ 1010\ (0A0006CA)\)
  - \(\text{BL label2} => 1110\ 101\ 1\ 0000\ 0000\ 0000\ 1000\ 1010\ 1110\ (EB0008AE)\)

Data Processing Instructions

- Uses 3-address format: first operand - always register; second operand - register/shifted register/immediate value; result - always a register
- For second register operand, shift can be logical/arithmetic/rotate. This is specified in "shift-type"
- How much to shift by is either a constant \#shift or a register
- For immediate value second operand, only rotation is possible
- S-bit controls condition code update
  - \(\text{N flag} - \text{set if result is negative (N equals bit 31 of result)}\)
  - \(\text{Z flag} - \text{set if result is zero}\)
  - \(\text{C flag} - \text{set if there is a carry-out from ALU during arithmetic operations, or set by shifter}\)
  - \(\text{V flag} - \text{set in an arithmetic operation if there is an overflow. It is significant only when operands are viewed as 2's complement signed values}\)
Data Processing Instruction Binary encoding

### ARM data processing instructions

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>AND</td>
<td>Logical bit-wise AND</td>
<td>Rd := Rn AND Op2</td>
</tr>
<tr>
<td>0001</td>
<td>EOR</td>
<td>Logical bit-wise exclusive OR</td>
<td>Rd := Rn EOR Op2</td>
</tr>
<tr>
<td>0010</td>
<td>SUB</td>
<td>Subtract</td>
<td>Rd := Rn - Op2</td>
</tr>
<tr>
<td>0011</td>
<td>RSB</td>
<td>Reverse subtract</td>
<td>Rd := Op2 - Rn</td>
</tr>
<tr>
<td>0100</td>
<td>ADD</td>
<td>Add</td>
<td>Rd := Rn + Op2</td>
</tr>
<tr>
<td>0101</td>
<td>ADC</td>
<td>Add with carry</td>
<td>Rd := Rn + Op2 + C</td>
</tr>
<tr>
<td>0110</td>
<td>SBC</td>
<td>Subtract with carry</td>
<td>Rd := Rn - Op2 + C - 1</td>
</tr>
<tr>
<td>0111</td>
<td>RSC</td>
<td>Reverse subtract with carry</td>
<td>Rd := Op2 - Rn + C - 1</td>
</tr>
<tr>
<td>1000</td>
<td>TST</td>
<td>Test</td>
<td>Scc on Rn AND Op2</td>
</tr>
<tr>
<td>1001</td>
<td>TEQ</td>
<td>Test equivalence</td>
<td>Scc on Rn EOR Op2</td>
</tr>
<tr>
<td>1010</td>
<td>CMP</td>
<td>Compare</td>
<td>Scc on Rn - Op2</td>
</tr>
<tr>
<td>1011</td>
<td>CMN</td>
<td>Compare negated</td>
<td>Scc on Rn + Op2</td>
</tr>
<tr>
<td>1100</td>
<td>ORR</td>
<td>Logical bit-wise OR</td>
<td>Rd := Rn OR Op2</td>
</tr>
<tr>
<td>1101</td>
<td>MOV</td>
<td>Move</td>
<td>Rd := Op2</td>
</tr>
<tr>
<td>1110</td>
<td>BIC</td>
<td>Bit clear</td>
<td>Rd := Rn AND NOT Op2</td>
</tr>
<tr>
<td>1111</td>
<td>MVN</td>
<td>Move negated</td>
<td>Rd := NOT Op2</td>
</tr>
</tbody>
</table>
Data processing instructions

- There are various different shift possibilities. The main ones:
  - 00 => LSL/ASL
  - 01 => LSR
  - 10 => ASR
  - 11 => ROR
- For “no shift”, e.g. ADD r0, r1, r2, assembler uses “logical shift left by zero”, e.g. ADD r0, r1, r2, LSL #0
- Note that for immediate operands (bit 25=1), rotation is the only allowed form of shift
  - a clever assembler may accept invalid instructions like ADD r0, r1, #1 LSL #31 and convert them to the acceptable ADD r0, r1, #1 ROR #1

Example of data processing instructions

- ADD r5, r1, r3
  - E081 5003
- ADDNES r0, r0, r0 LSL #2
  - 1090 0100
Data Transfer Instructions (LDR/STR)

- Address of load / store is base register value +/- <offset>

- P = 1 means pre-indexed, e.g. LDR r0, [r1,#4]
- P = 0 means post-indexed, e.g. LDR r0, [r1], #4
- B = 1 selects byte transfer (B = 0 is word transfer)
- W = 1 is “write-back”, e.g. LDR r0, [r1,#4]! (! => W=1) [should usually have W=0 if P=0]
- <offset> may be 12-bit unsigned immediate value (constant)
- <offset> may also be a shifted register
- the sign of the offset is determined by U. U=1 => add this offset, U=0 => subtract this offset
- L=1 means “load”, L=0 means “store”
- All the shift parameters are the same as before
Data Transfer instructions

- All loads and stores are relative to a base register
  - fine for STR r2, [r1]
  - what about STR r2, label?
- Solution is to calculate the address of "label" relative to the current value of the PC, and then use the PC as the base register
- STR r2, label
  - say this instruction is at address 0xF8
  - say "label" corresponds to address 0x1FF
  - as with branch instructions, value of PC will be 0xF8 + 8 = 0x100 when executing
  - offset is then 0x1FF – 0x100 = 0xFF
  - so use STR r2, [PC, #0xFF] (E58F20FF)
- Thankfully, the assembler does this work for us – we can just write STR r2, label. The assembler translates this to a PC-relative store and calculates the offset

Multiple register transfer instructions

- Register list has one bit per register
- bit 0 = 1 => load/store r0; bit 1 = 1 => load/store r1; etc
- STMIA r13!, {r0-r2, r14} E8AD 4007
SWI instructions

- We have used SWI (software interrupt) to communicate with external devices, e.g. the monitor
- More detail on what exactly an SWI is, and the variety of SWIs available, later this course
- For now, we will concentrate on the instruction format

- For example, SWI 0x0, used in our “Hello World!” example program
- SWI 0x0 => EF000000

Multiply Instructions

- ARM has a number of multiply instructions
  - Produces the product of two 32-bit binary numbers held in registers.
  - Result of 32-bit*32-bit is 64 bits. Some ARM processors store the entire 64-bit result in registers. Other ARM processors only store the LOWER 32-bit of the product.
  - A Multiply-Accumulate instruction also adds the product to the accumulator value to form a running total.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>MUL</td>
<td>Multiply (32-bit result)</td>
<td>Rd := (Rm * Rs) [31:0]</td>
</tr>
<tr>
<td>001</td>
<td>MLA</td>
<td>Multiply-accumulate (32-bit result)</td>
<td>Rd := (Rm * Rs + Rn) [31:0]</td>
</tr>
<tr>
<td>100</td>
<td>UMLA</td>
<td>Unsigned multiply long</td>
<td>RdHi:RdLo := Rm * Rs</td>
</tr>
<tr>
<td>101</td>
<td>UMLAL</td>
<td>Unsigned multiply-accumulate long</td>
<td>RdHi:RdLo += Rm * Rs</td>
</tr>
<tr>
<td>110</td>
<td>SMULL</td>
<td>Signed multiply long</td>
<td>RdHi:RdLo := Rm * Rs</td>
</tr>
<tr>
<td>111</td>
<td>SMLAL</td>
<td>Signed multiply-accumulate long</td>
<td>RdHi:RdLo += Rm * Rs</td>
</tr>
</tbody>
</table>
Example of using ARM Multiplier

- This calculates a scalar product of two vectors, 20 long.
- r8 and r9 points to the two vectors
- r11 is the loop counter
- r10 stores the result

```asm
MOV r11, #20 ; initialize loop counter
MOV r10, #0 ; initialize total
LOOP
  LDR r0, [r8], #4 ; get first component
  LDR r1, [r9], #4 ; ... and second
  MLA r10, r0, r1, r10 ; accumulate product
  SUBS r11, r11, #1 ; decrement loop counter
BNE LOOP
```

Multiplication by a constant

- When multiplying by a constant value, it is possible to replace the general multiply with a fixed sequence of adds and subtractions that have the same effect.
- For instance, multiply by 5 could be achieved using a single instruction:

```asm
ADD Rd, Rm, Rm, LSL #2 ; Rd = Rm + (Rm * 4) = Rm * 5
```

- This is obviously better than the MUL version:

```asm
MOV Rs, #5
MUL Rd, Rm, Rs
```

- What constant multiplication is this?

```asm
ADD r0, r0, r0, LSL #2 ; r0' := 5 x r0
RSB r0, r0, r0, LSL #3 ; r0'' := 7 x r0'
```