Microprocessors

6. ARM assembler

Stanislav Vítek
Katedra radioelektroniky
České vysoké učení technické v Praze

ARM Programmer-visible Registers

ARM implements sixteen 32-bit processor registers labeled R0 through R15.

- R15 is the program counter (PC)
- R14 is the link register (LR)
- R13 is the stack pointer (SP)

In general, we use only∗ R0...R12 as General Purpose Registers (GPRs) and only use and refer to R13, R14, and R15 as SP, LR, and PC.

- In practice, additional guidelines further limit the use of registers by programmers and compilers. Curious? See the ARM Architecture Procedure Call Standard.

There is also a special status register called the Current Program Status Register (CPSR) that indicates various useful information (more later).

Assembly Language Syntax

Assembly language consists of shorthand instruction names called mnemonics, a syntax for using them, and other directives for organizing them.

A program called an assembler translates the mnemonics into machine language instructions (binary; more later).

Here is a (short) ARM assembly program:

```
ADD R1, R2, R3 // R1 <-- R2 + R3
```

- ADD is a mnemonic
- R1 is a destination register; the first operand
- R2 and R3 are source registers; the second and third operand
- // R1 <-- R2 + R3 is a comment (not a very useful one)

There are different ways to use each instruction.

```
ADD R1, R2, R3 // R1 <-- R2 + R3
```

Here, the syntax of the instruction is ADD Rd, Rn, Rm where

- Rd specifies the destination register
- Rn and Rm specify the source registers

```
ADD R4, R5, #24 // R4 <-- R5 + 24
```

Here, the syntax of the instruction is ADD Rd, Rn, Imm where

- Rd specifies the destination register
- Rn specifies the source register
- Imm specifies an immediate value (constant)

Move Instructions

These instructions copy data into registers from other registers or immediate values.

```
MOV Rd, Op2      // MOVes value of Op2 into Rd
MOV Rd, #Imm16   // MOVes immediate 16-bit value into Rd
MVN Rd, Op2      // MOVes complement (Not) of Op2 value into Rd
MOVT Rd, #Imm16 // MOVes Top: moves a 16-bit constant into
                // the high-order 16 bits of Rd and leaves
                // the lower bits unchanged
```

Logic Instructions

These instructions perform binary logic operations on operands, useful for testing conditions, manipulating data, etc.

```
AND Rd, Rn, Op2 // bitwise AND operation
ORR Rd, Rn, Op2 // bitwise OR operation
EOR Rd, Rn, Op2 // bitwise Exclusive OR (XOR) operation
BIC Rd, Rn, Op2 // BIt Clear: Rd <-- Rn AND NOT(Op2)
```

Shift and Rotate Instructions

Shift and rotate instructions change the positions of bits within a register, moving them left or right.

Note: Last operand can be a register or an immediate value, as with logic operations.

```
LSL R1, R2, #5 // Logical shift left
LSR R1, R2, R3 // Logical shift right
ASR R1, R2, #4 // Arithmetic shift right
```

- Logical ⇒ pad with 0s, Arithmetic ⇒ extend sign bit

```
ROR R1, R2, #2 // Circular rotate right
```

- Less significant bits (on the right of the register) are moved into the most significant positions (on the left of the register).

Arithmetic Instructions

Addition/subtraction instructions:

```
ADD R0, R1, R2 // R0 <-- R1 + R2
ADD R0, R1, #-24 // R0 <-- R1 + (-24)
SUB R0, R1, #24 // R0 <-- R1 - (24)
ADD R0, R1, R2, LSL#2 // R0 <-- R1 + R2*4
```

Multiply instruction

```
MUL R2, R3, R4 // R2 <-- R3 * R4
Multiply-accumulate instruction
MLA R2, R3, R4, R5 // R2 <-- (R3 * R4) + R5
```

These multiply instructions only return the 32 least significant bits.

Arithmetic Instructions

What about division?

```
UDIV R0, R1, R2 // R0 <-- R1 / R2, R1 and R2 unsigned
SDIV R0, R1, R2 // R0 <-- R1 / R2, R1 and R2 signed
```

But many processors do not implement division.

Division hardware is

- Complex, and therefore costly;
- Slow; and,
- Used infrequently.

Consequently, it is often performed in software using an ARM-provided library subroutine (e.g., aeabi_idiv()).

Array Access

Consider the following C code snippet:

```c
int arr[8] = {17, 58, 79, 15, ...} ; // sizeof (int) = 4B

for (int i = 0; i < 8; i++ ) {
  v = arr[i] ;
  // ---
  arr[i] = v ;
}
```

When reading from an array, we need to:

- Get the base address (&arr);
- Multiply the index by the element size (i*4) to get the offset;
- Add to calculate the address of the element; and, then, finally
- Access memory!

To access arr we need an instruction that can read from memory:

```
LDR Rd, [Rn] // Rd <-- Mem[Rn], Rn = address in bytes
```

Our C code is implemented in part with the following assembly:

```
// R0 = variable i
// R1 = base address of arr (&arr)
MOV R2, #4     // R2 = 4
MUL R2, R0, R2 // R2 = i*4 -- calculate offset for index i
ADD R3, R1, R2 // R3 = arr + i*4 -- absolute address of arr[i]
LDR R4, [R3]   // R4 = arr[i] -- R4 <-- Mem[R1+i*4]
```

Load and Store Instructions

Memory accesses commonly* access words and take the form of:

```
LDR Rd, <EA> // Rd <-- Mem[EA]; reads a 32-bit word
STR Rm, <EA> // Mem[EA] <-- Rn; writes a 32-bit word
```

Loads and stores do not generally specify a memory address explicitly; instead, they compute an effective address (EA) from a base address and an offset.

Effective Address Calculation

EA = base + offset

Calculating an EA is very convenient for implementing common program structures: e.g., loops and arrays; and, complex objects.

- Other load and store instructions access bytes or half words, doubles, or multiple words, and manipulate addresses in more complex ways.

Effective Address Calculation

The base address is always stored in a register (Rn). There are three kinds of offset:

- Immediate: a 12-bit number added to or subtracted from the base address
- Index register: the offset is stored in a register (Rm)
- Scaled index register: the value in the index register is shifted by a specified immediate value, then added to or subtracted from the base address

Name	Assembler syntax	Address generation
register indirect	[Rn]	EA = Rn
immediate offset	[Rn, #offset]	EA = Rn + offset
offset in Rm	[Rn, ± Rm, shift]	EA = Rn ± shifted(Rm)