ARMv4 Data Processing Instructions

Instructor: Dr. Vinicius Prado da Fonseca (vpradodafons@online.mun.ca)

ARMv4 Instructions

Unit 5 outcomes:

- RTL (R0 <= R0 + R1) description to Assembly language
- Assembly language to machine code
- Machine code to the datapath

We saw a small version of this “workflow” in Unit 4 with the mini datapath.

Three main types:

- Data processing instructions (this lecture)
- Memory access instructions
- Branch instructions

Data Processing Instructions (register)

- Logical and arithmetic instructions
- Instructions with register operands have the following form:

```
OP Rd, Rn, Rm
```

Instruction	RTL
AND Rd, Rn, Rm	R[d] <= R[n] & R[m]
EOR Rd, Rn, Rm	R[d] <= R[n] ^ R[m]
SUB Rd, Rn, Rm	R[d] <= R[n] - R[m]
RSB Rd, Rn, Rm	R[d] <= R[m] - R[n]
ADD Rd, Rn, Rm	R[d] <= R[n] + R[m]
ADC Rd, Rn, Rm	R[d] <= R[n] + R[m] + C
SBC Rd, Rn, Rm	R[d] <= R[n] - R[m] - ~C
RSC Rd, Rn, Rm	R[d] <= R[m] - R[n] - ~C
TST Rd, Rn, Rm	R[m] & R[n], set codes
TEQ Rd, Rn, Rm	R[m] ^ R[n], set codes
CMP Rd, Rn, Rm	R[n]-R[m], set codes
CMN Rd, Rn, Rm	R[n] + R[m], set codes
ORR Rd, Rn, Rm	R[d] <= R[n] | R[m]
MOV Rd, Rm	R[d] <= R[m]
BIC Rd, Rn, Rm	R[d] <= R[n] & ~R[m]
MVN Rd, Rn, Rm	R[d] <= ~R[m]

Data Processing Instructions (register)

```
OP Rd, Rn, Rm
```

- OP can be:
  - AND
  - EOR
  - SUB
  - RSB, reverse subtraction
  - ADD
  - ADC, add register plus carry-in
  - SBC, sub minus complement carry-in
  - RSC, reverse, sub minus complement carry-in
  - TST, bitwise AND, set the flags
  - TEQ, bitwise EOR, set the flags
  - CMP, sub, set the flags
  - CMN, add, set the flags
  - ORR
  - MOV, copy values
  - BIC, bitwise AND, Rm complement
  - MVN, complement

Instruction	RTL
AND Rd, Rn, Rm	R[d] <= R[n] & R[m]
EOR Rd, Rn, Rm	R[d] <= R[n] ^ R[m]
SUB Rd, Rn, Rm	R[d] <= R[n] - R[m]
RSB Rd, Rn, Rm	R[d] <= R[m] - R[n]
ADD Rd, Rn, Rm	R[d] <= R[n] + R[m]
ADC Rd, Rn, Rm	R[d] <= R[n] + R[m] + C
SBC Rd, Rn, Rm	R[d] <= R[n] - R[m] - ~C
RSC Rd, Rn, Rm	R[d] <= R[m] - R[n] - ~C
TST Rd, Rn, Rm	R[m] & R[n], set codes
TEQ Rd, Rn, Rm	R[m] ^ R[n], set codes
CMP Rd, Rn, Rm	R[m] - R[n], set codes
CMN Rd, Rn, Rm	R[n] + R[m], set codes
ORR Rd, Rn, Rm	R[d] <= R[n] | R[m]
MOV Rd, Rm	R[d] <= R[m]
BIC Rd, Rn, Rm	R[d] <= R[n] & ~R[m]
MVN Rd, Rn, Rm	R[d] <= ~R[m]

Data Processing Instructions (register)

- Rd, Rn, Rm are registers operands
- Rd is the destination
- Rn and Rm are sources
- d, n, and m range between 0 and 15 (4 bits)
  - Fixed number of registers
- Immediate operands are also available, discussed later
- Table with data processing instructions used in this course

Instruction	RTL
AND Rd, Rn, Rm	R[d] <= R[n] & R[m]
EOR Rd, Rn, Rm	R[d] <= R[n] ^ R[m]
SUB Rd, Rn, Rm	R[d] <= R[n] - R[m]
RSB Rd, Rn, Rm	R[d] <= R[m] - R[n]
ADD Rd, Rn, Rm	R[d] <= R[n] + R[m]
ADC Rd, Rn, Rm	R[d] <= R[n] + R[m] + C
SBC Rd, Rn, Rm	R[d] <= R[n] - R[m] - ~C
RSC Rd, Rn, Rm	R[d] <= R[m] - R[n] - ~C
TST Rd, Rn, Rm	R[m] & R[n], set codes
TEQ Rd, Rn, Rm	R[m] ^ R[n], set codes
CMP Rd, Rn, Rm	R[m] - R[n], set codes
CMN Rd, Rn, Rm	R[n] + R[m], set codes
ORR Rd, Rn, Rm	R[d] <= R[n] | R[m]
MOV Rd, Rm	R[d] <= R[m]
BIC Rd, Rn, Rm	R[d] <= R[n] & ~R[m]
MVN Rd, Rn, Rm	R[d] <= ~R[m]

Data Processing Instructions (register)

- Register instruction examples

```
mov r1, r2
add r0, r0, r1
```

What is the RTL description for these instructions?

Instruction	RTL
AND Rd, Rn, Rm	R[d] <= R[n] & R[m]
EOR Rd, Rn, Rm	R[d] <= R[n] ^ R[m]
SUB Rd, Rn, Rm	R[d] <= R[n] - R[m]
RSB Rd, Rn, Rm	R[d] <= R[m] - R[n]
ADD Rd, Rn, Rm	R[d] <= R[n] + R[m]
ADC Rd, Rn, Rm	R[d] <= R[n] + R[m] + C
SBC Rd, Rn, Rm	R[d] <= R[n] - R[m] - ~C
RSC Rd, Rn, Rm	R[d] <= R[m] - R[n] - ~C
TST Rd, Rn, Rm	R[m] & R[n], set codes
TEQ Rd, Rn, Rm	R[m] ^ R[n], set codes
CMP Rd, Rn, Rm	R[m] - R[n], set codes
CMN Rd, Rn, Rm	R[n] + R[m], set codes
ORR Rd, Rn, Rm	R[d] <= R[n] | R[m]
MOV Rd, Rm	R[d] <= R[m]
BIC Rd, Rn, Rm	R[d] <= R[n] & ~R[m]
MVN Rd, Rn, Rm	R[d] <= ~R[m]

Data Processing Instructions (register)

- Register instruction examples

```
mov r1, r2
add r0, r0, r1
```

What is the RTL description for these instructions?

R[1] <= R[2]

R[0] <= R[0] + R[1]

Instruction	RTL
AND Rd, Rn, Rm	R[d] <= R[n] & R[m]
EOR Rd, Rn, Rm	R[d] <= R[n] ^ R[m]
SUB Rd, Rn, Rm	R[d] <= R[n] - R[m]
RSB Rd, Rn, Rm	R[d] <= R[m] - R[n]
ADD Rd, Rn, Rm	R[d] <= R[n] + R[m]
ADC Rd, Rn, Rm	R[d] <= R[n] + R[m] + C
SBC Rd, Rn, Rm	R[d] <= R[n] - R[m] - ~C
RSC Rd, Rn, Rm	R[d] <= R[m] - R[n] - ~C
TST Rd, Rn, Rm	R[m] & R[n], set codes
TEQ Rd, Rn, Rm	R[m] ^ R[n], set codes
CMP Rd, Rn, Rm	R[m] - R[n], set codes
CMN Rd, Rn, Rm	R[n] + R[m], set codes
ORR Rd, Rn, Rm	R[d] <= R[n] | R[m]
MOV Rd, Rm	R[d] <= R[m]
BIC Rd, Rn, Rm	R[d] <= R[n] & ~R[m]
MVN Rd, Rn, Rm	R[d] <= ~R[m]

Bit Shifting Instructions

- ARMv4 supports four types of bit shifting instructions.
  - Left Shift
    - moves bits from their current bit position to a position a fixed number of bits to the left
    - The shift amount can vary from 1 to the word size
    - Zero is shifted in to position 0
    - In Verilog the << operator performs a left shift
    - Shifting left is equivalent to multiplication by a power of two
    - One shift left is the same as multiplication by two
  - Logic Right Shift
    - A right shift moves bits from their current bit position to a position a fixed number of bits to the right
    - The shift amount can vary from 1 to the word size
    - In Verilog the >> operator performs a right shift
    - A zero is shifted into the most significant bit position
    - Right shifting is equivalent to dividing by a power of two
    - One right shift divides by two

Bit Shifting Instructions

- ARMv4 supports four types of bit shifting instructions.
  - Arithmetic Right Shift
    - The arithmetic right shift is the same as the logical right shift, except the sign bit is preserved
    - Its Verilog syntax is: >>>
    - Arithmetic right shift is used for signed numbers.
  - Rotate Right
    - The bits are rotated right, the bit rotate from the least significant bit are shifted into the most significant bit
    - Rotate shift are also called circular shifts.

Bit Shifting Instructions

- The shifting instructions are: LSR, LSL, ASR, ROR, and RRX
  - The Rd operand specifies the destination
  - Rm is the source register to shift
  - Rs is the register holding a shift amount. Immediate #sh is a constant shift can also be used.
- RRX creates a 33 bit value to rotate using the C bit
- Their RTL description is:

Instruction	RTL
LSR Rd, Rm, Rs	R[d] <= R[m] >> R[s]
LSL Rd, Rm, Rs	R[d] <= R[m] << R[s]
ASR Rd, Rm, Rs	R[d] <= R[m] >>> R[s]
ROR Rd, Rm, Rs	R[d] <= {R[n][sh-1:0],R[n][31:sh]} where sh = R[s]
RRX Rd, Rm	R[d] = {C,R[d]} <= {R[m][0],C,R[m][31:1]}

Data Processing Instructions (immediate)

- The ARMv4 architecture can replace Rm with a 12-bit immediate constant.
  - The format is:

```
OP Rd, Rn, #immediate
```

- OP is the same as the data processing and shifting instructions
- The 12 bits represent all 32-bit possible numbers
  - 4 bits specify a right rotate
  - 8 bits are zero extended to 32 bits and rotate
- The constant will be explained in more detail later

Data Processing Instructions (immediate)

- Operations and shift with immediate:

```
mov r1, #5
add r0, r1, #7
lsl R1, R1, #1
```

Instruction	RTL
AND Rd, Rn, #immed	R[d] <= R[n] & immed
EOR Rd, Rn, #immed	R[d] <= R[n] ^ immed
SUB Rd, Rn, #immed	R[d] <= R[n] - immed
RSB Rd, Rn, #immed	R[d] <= immed - R[n]
ADD Rd, Rn, #immed	R[d] <= R[n] + immed
ADC Rd, Rn, #immed	R[d] <= R[n] + immed + C
SBC Rd, Rn, #immed	R[d] <= R[n] - immed - ~C
RSC Rd, Rn, #immed	R[d] <= immed - R[n] - ~C
TST Rd, Rn, #immed	immed & R[n], set codes
TEQ Rd, Rn, #immed	immed ^ R[n], set codes
CMP Rd, Rn, #immed	immed - R[n], set codes
CMN Rd, Rn, #immed	R[n] + immed, set codes
ORR Rd, Rn, #immed	R[d] <= R[n] | immed
MOV Rd, #immed	R[d] <= immed
BIC Rd, Rn, #immed	R[d] <= R[n] & ~immed
MVN Rd, Rn, #immed	R[d] <= ~immed
LSR Rd, Rm, #immed	R[d] <= R[m] >> immed
LSL Rd, Rm, #immed	R[d] <= R[m] << immed
ASR Rd, Rm, #immed	R[d] <= R[m] >>> immed
ROR Rd, Rm, #immed	R[d] <= {R[n][sh-1:0],R[n][31:sh]} where sh = immed

ARMv4 Assembly Language (gnu)

- The assembler program
  - Translate a symbolic representation of instructions and data
  - Into the ones and zeros understood by a machine
- The following examples uses the gnu assembler program:

```
arm-linux-gnueabi-as
```

- Manual and online tutorial link in the notes
- GNU toolchain

ARMv4 Assembly Language (gnu)

- Small program that adds 5 and 7
- This program uses the exit status to return the sum

```
int main() {
     int r0 = 5;
     int r1 = 7;

     r0 = r0 + r1;

     return r0;
}
```

```
     @ a comment
     .arch armv4
     .text
     .align   2
     .global  main
     .arm
main:
     mov r0, #5
     mov r1, #7
     add r0, r0, r1
     mov pc, lr
     .size main, .-main
```

ARMv4 Assembly Language (gnu)

- Comments:
  - @
  - //
  - /* */

```
        @ a comment
        .arch armv4
        .text
        .align   2
        .global  main
        .arm
main:
        mov r0, #5
        mov r1, #7
        add r0, r0, r1
        mov pc, lr
        .size main, .-main
```

ARMv4 Assembly Language (gnu)

- Words starting with a " . " are directives to the assembler.
  - .arch armv4
    - states that the assembler code is from the armv4 ISA
  - .text
    - An assembler deals with instructions and data
    - The .text directive starts a text (instruction) section
  - .align 2
    - ARM keeps current memory address
    - the lower two bits are made zero, by rounding the current memory address

```
        @ a comment
        .arch armv4
        .text
        .align   2
        .global  main
        .arm
main:
        mov r0, #5
        mov r1, #7
        add r0, r0, r1
        mov pc, lr
        .size main, .-main
```

ARMv4 Assembly Language (gnu)

- .global main (public in Java)
- .arm

```
      @ a comment
      .arch armv4
      .text
      .align   2
      .global  main
      .arm
main:
      mov r0, #5
      mov r1, #7
      add r0, r0, r1
      mov pc, lr
      .size main, .-main
```

ARMv4 Assembly Language (gnu)

- Assembler deals with instructions and data
- Instruction and data are placed in different sections of memory

```
      @ a comment
      .arch armv4
      .text
      .align   2
      .global  main
      .arm
main:
      mov r0, #5
      mov r1, #7
      add r0, r0, r1
      mov pc, lr
      .size main, .-main
```

ARMv4 Assembly Language (gnu)

- The assembler program maintains a current memory address where the next instruction or data will be placed
- ".align 2" ensures that address is a multiple of 4
- Java, C global

```
      @ a comment
      .arch armv4
      .text
      .align   2
      .global  main
      .arm
main:
      mov r0, #5
      mov r1, #7
      add r0, r0, r1
      mov pc, lr
      .size main, .-main
```

ARMv4 Assembly Language (gnu)

- The current location can be assigned to a symbolic name
  - an identifier followed by a “ : ” (colon)
  - label
- Example:
  - main is a label for the current location, PC = 0

```
        @ a comment
        .arch armv4
        .text
        .align   2
        .global  main
        .arm
main:
        mov r0, #5
        mov r1, #7
        add r0, r0, r1
        mov pc, lr
        .size main, .-main
```

ARMv4 Assembly Language (gnu)

mov r0, #5

- The assembler generates the mov instruction binary encoding
- Encoding includes the operands
- Execution of mov copies the 12-bit value stored in the instruction to the register
- r0 will contain 5
- Since all ARMv4 instructions are 32 bits (4 bytes), the location PC will now be PC+4

```
        @ a comment
        .arch armv4
        .text
        .align   2
        .global  main
        .arm
main:
        mov r0, #5
        mov r1, #7
        add r0, r0, r1
        mov pc, lr
        .size main, .-main
```

ARMv4 Assembly Language (gnu)

mov r1, #7

- After execution, r1 will contain 7
- The location is advanced to 8.

add r0, r0, r1

- An instruction to add r0 and r1, storing the result in r0 is encoded
- Again, the location is advanced by 4 to be 12 (C_{16}).

```
        @ a comment
        .arch armv4
        .text
        .align   2
        .global  main
        .arm
main:
        mov r0, #5
        mov r1, #7
        add r0, r0, r1
        mov pc, lr
        .size main, .-main
```

ARMv4 Assembly Language (gnu)

mov pc, lr

- Return to the code that will exit the program
- lr has the last branch address

0100 b main
0104 ⇐ saved to lr

.size main, .-main

- current location “ . ”
- “ . -main” size in bytes for this code sequence

```
        @ a comment
        .arch armv4
        .text
        .align   2
        .global  main
        .arm
main:
        mov r0, #5
        mov r1, #7
        add r0, r0, r1
        mov pc, lr
        .size main, .-main
```

Assembling add.s

- Invoking the assembler to create the machine code from add.s

```
arm-linux-gnueabi-as -al add.s
```

- “-al” options asks the assembler to produce a listing
- Shows how the source was assembled

Assembling add.s

- First group of four hexadecimal digits shows the memory location
- Next group of eight hexadecimal digits is the encoded instructions
- Rest of the line shows the source line that generated the encoded instruction.

```
7                       main:

8   0000 0500A0E3       mov r0, #5

9   0004 0710A0E3       mov r1, #7

10  0008 010080E0       add r0, r0, r1

11  000c 0EF0A0E1       mov pc, lr

12                      .size main, .-main
```

Assembling add.s

- Big-endian "big end" (most significant value in the sequence)
  - is stored first, at the lowest storage address.
- Little-endian "little end" (least significant value in the sequence)
  - is stored first.
- Compiler stores instructions in big-endian
- Our examples will always use little-endian
- 05 is the least significant byte of the instruction (bits 7:0).
- The most significant byte is E3 (bits 31:24)

```
0500A0E3 -> E3A00005

0710A0E3 -> E3A01007
```

```
7                       main:

8   0000 0500A0E3       mov r0, #5

9   0004 0710A0E3       mov r1, #7

10  0008 010080E0       add r0, r0, r1

11  000c 0EF0A0E1       mov pc, lr

12                      .size main, .-main
```

Assembling add.s

mov r0, #5

0xE3A00005

0b11100011101000000000000000000101

mov r0, #7

0xE3A01007

0b11100011101000000001000000000111

- Hand assembly, is seen in detail later

```
7                       main:
8   0000 0500A0E3       mov r0, #5
9   0004 0710A0E3       mov r1, #7
10  0008 010080E0       add r0, r0, r1
11  000c 0EF0A0E1       mov pc, lr
12                      .size main, .-main
```

ADD Example

- The assembly file is translated to create an object file
- Contains the encoded data and instructions
- Also contains a symbol table containing labels and locations.

```
arm-linux-gnueabi-as add.s -o add.o  # -o specified the output

arm-linux-gnueabi-nm -g add.o # -g displays only globals

arm-linux-gnueabi-objdump -S add.o # disassembled

rm -f add.o   # clean up files

arm_emu.sh also can be used
```

Executing add.s with qemu-arm

- An executable ARM program is created and run with the commands

```
arm-linux-gnueabi-gcc --static add.s

qemu-arm ./a.out ; echo $?
```

- --static option creates an executable with no dynamic libraries
- Return value of the main program is displayed with echo $?

- The emulator is required when using a PC that uses the x86 ISA
- The program could be executed directly on a Raspberry Pi.

Multiply By 19

- Code to multiply 19

```
int mult19(int value) {
     int result = value * 19;

     return result;
}
```

- We will see function parameters later

Multiply By 19

An expression to multiply a value by 19 is:

$a << 4 + a + a + a = 16*a + 3*a = 19*a$

- mult19.s
- mult19v1.s

```
int mult19(int value) {
     int result = value * 19;

     return result;
}
```

- We will see function parameters later

mult19.s

```
main:
      // multiply r0 by 19
      // a << 4 + a + a + a = 16*a + 3*a = 19*a
      mov  r0, #10        // r0 = 10, will be passed as parameter
      mov  r1, r0         // save orginal value of r0
      lsl  r0, r0, #4     // r0 = r0 * 16
      add  r0, r0, r1     // r0 = r0 + r1
      add  r0, r0, r1     // r0 = r0 + r1
      add  r0, r0, r1     // r0 = r0 + r1
      mov  pc, lr         // return value in r0
      .size main, .-main
```

mult19v1.s

```
main:
      // multiply r0 by 19
      mov  r0, #10              // r0 = 10
      lsl  r1, r0, #4           // r1 = r0 * 16
      add  r1, r1, r0, lsl #1   // r1 = r1 + r0 * 2
      add  r0, r0, r1           // r0 = r0 + r1
      mov  pc, lr         // return value in r0
      .size main, .-main
```

Expression Evaluation

- An assembly language program that evaluates the expression:

  (a - 2*b) + (c - d)

- expr.s

```
int main(int a, int b, int c, int d) {
        int result = (a - 2*b) + (c - d);

        return result;
}
```

- We will see function parameters later

expr.s

```
main:
        // expression for (a - 2*b) + (c-d)
        // r0 is a, r1 is b, r2 is c, r3 is d
        mov  r0, #14        // a = 14
        mov  r1, #3         // b = 3
        mov  r2, #22        // c = 22
        mov  r3, #10        // d = 10
        lsl  r1, r1, #1     // b = 2 * b
        sub  r0, r0, r1     // a = a - b
        sub  r2, r2, r3     // c = c - d
        add  r0, r0, r2     // a = a + c
        mov  pc, lr         // return value in r0
        .size main, .-main
```

Questions?