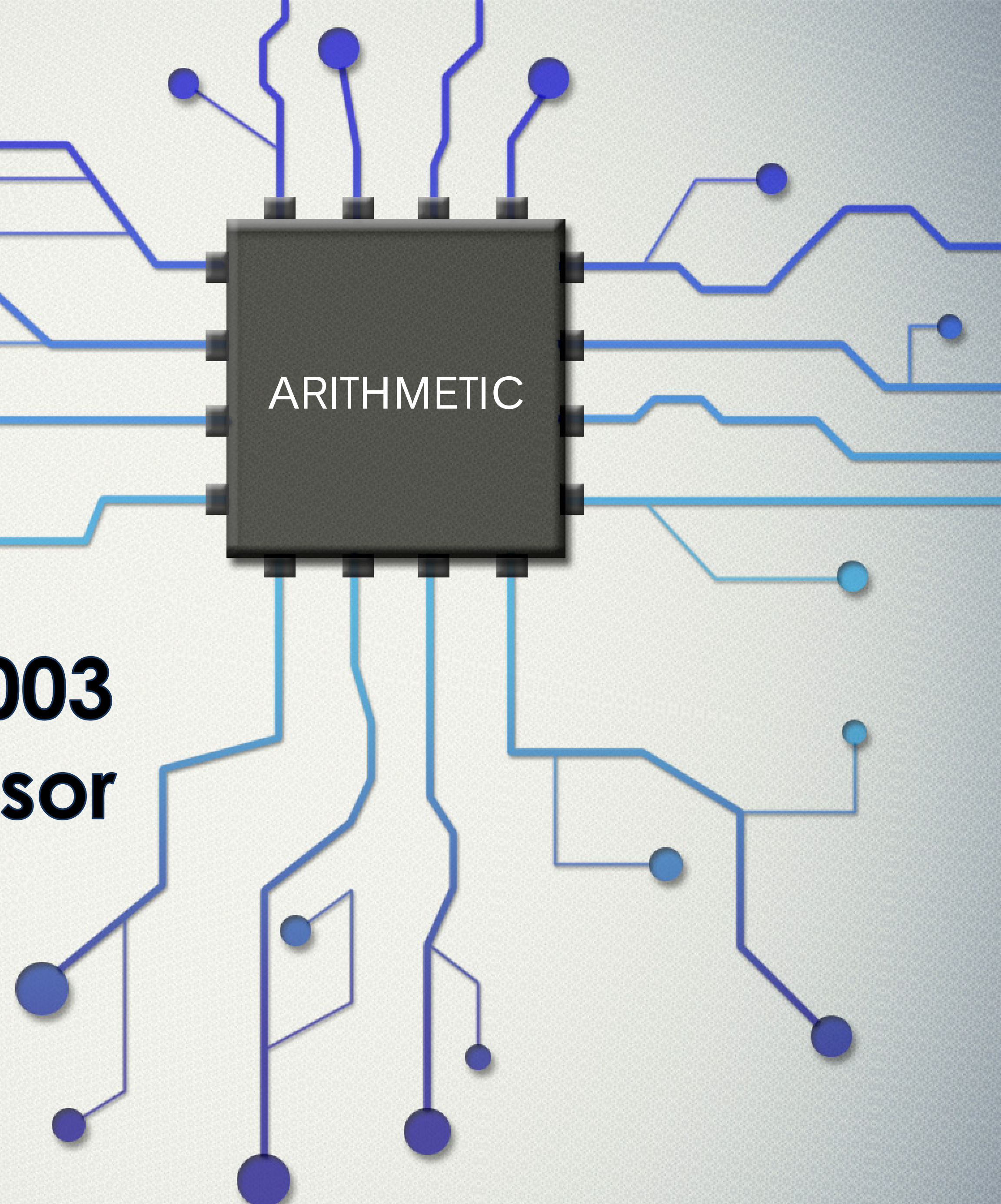
EE3002/ IM2003 Microprocessor Part 1



Chapter 7 Part 2

+−×/

- Engineers tend to use lots of mathematical operations in their programs.
- This lecture will look at how arithmetic are implemented in ARM assembly instructions

Addition/Subtraction

- Addition
  - ADD r1, r2, r3 ;r1 = r2 + r3
  - ADC r1, r2, r3 ;r1 = r2 + r3 + C
- Subtraction
  - SUB r1, r2, r3 ;r1 = r2 – r3
  - SBC r1, r2, r3 ;r1 = r2 – r3 + C -1
- Reverse Subtraction
  - RSB r1, r2, r3 ;r1 = r3 – r2
  - RSC r1, r2, r3 ;r1 = r3 – r2 + C -1

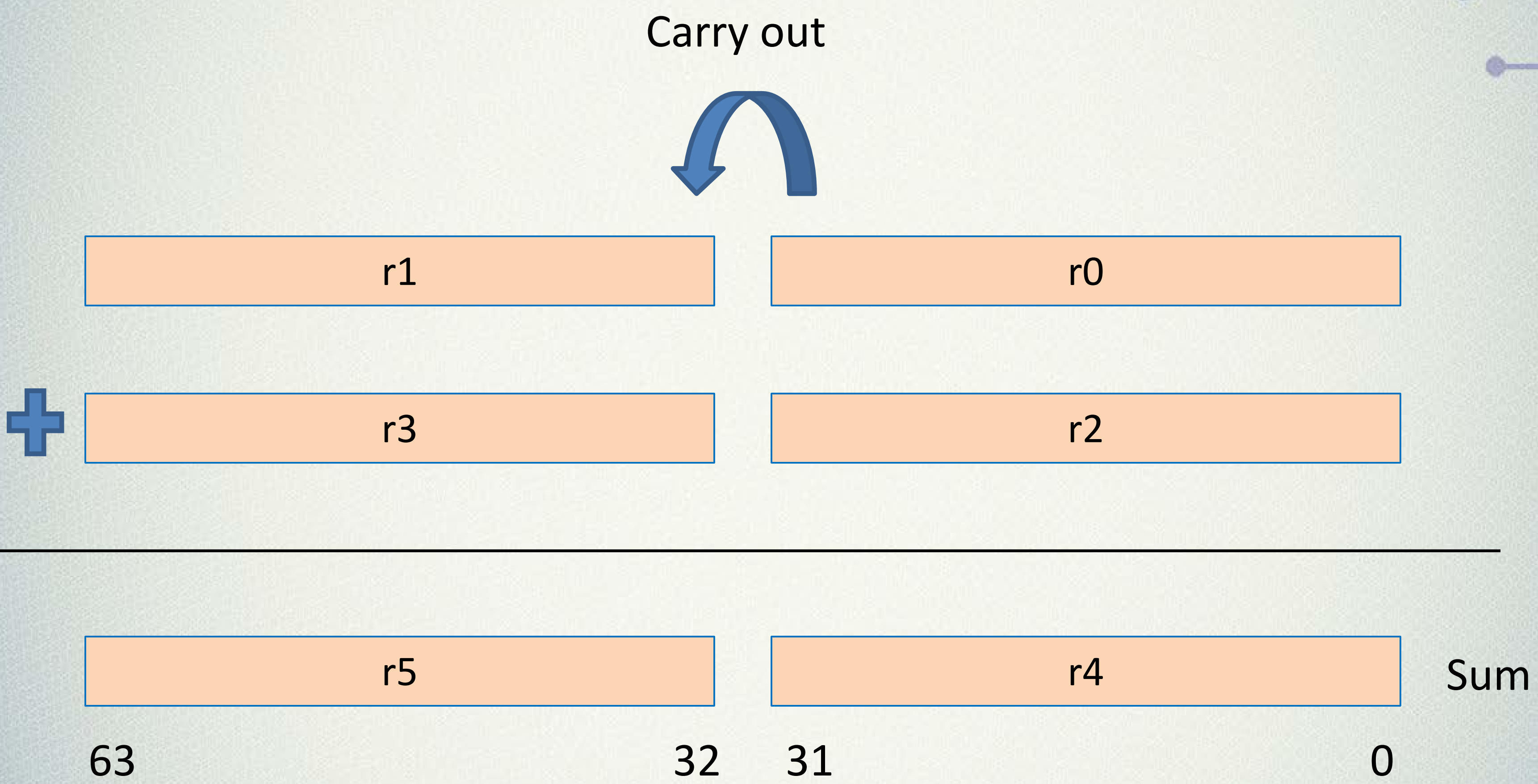
64-bit addition

- The following instructions add a 64-bit integer contained in r2 and r3 to another 64-bit integer in r0 and r1 and place the result in registers r4 and r5 :

ADDS r4, r0, r2 ; add the least significant word

ADC r5, r1, r3; add the most significant word

64-bit addition



64-bit subtraction

- A 64-bit subtraction can be built by first subtracting the lower halves of 64-bit values, updating the Carry flag, and then subtracts the upper halves, including the Carry:

SUBS r0, r0, r2 ;subtract lower halves, set ;Carry flag

SBCr1, r1, r3 ;subtract upper halves ;and Carry

Absolute value example

- Write ARM assembly to compute the absolute value
- r1 contains the initial value, r0 contains the absolute value.
- This can be done using the following two lines

```
CMP    r1, #0 ;check if r1 negative
RSBLT  r0, r1, #0 ; if r1 negative, r0 = 0-r1
```

Multiplication

Instruction	Comment
MUL	32x32 multiply with 32-bit product
MLA	32x32 multiply added to a 32-bit accumulated value
SMULL	Signed 32x32 multiply with 64-bit product
UMULL	Unsigned 32x32 multiply with 64-bit product
SMLAL	Signed 32x32 multiply added to a 64-bit accumulated value
UMLAL	Unsigned 32x32 multiply added to a 64-bit accumulated value

Multiplication examples

- MUL r4, r3, r1 ; r4 = r3 * r1
- MULS r4, r2, r1 ; r4 = r2 * r1, set the flags also
- MLA r7, r8, r9, r3;r7 = r8 * r9 + r3
- SMULL r4, r8, r2, r3 ;r4 = bits 31-0 of r2*r3

  ;r8 = bits 63-32 of r2*r3, i.e. r4 contains the least

  ;significant word, while r8 contains the most significant

  ;word
- UMULL r6, r8, r0, r1 ;{r6,r8} = r0*r1
- SMLAL r4, r8, r2, r3 ;{r4,r8} = r2*r3 + {r4,r8}
- UMLAL r5,r8, r0, r1 ;{r5,r8} = r0*r1 + {r5,r8}

Multiplication by a constant

- It is sometimes possible to perform multiplications without using the hardware multiplier which may consumes more power and time.
- This can be done by making clever use of the barrel shifter for operand 2.
- Multiplying a number by a power of two can be easily perform using the barrel shifter

  MOV r1, r0, LSL #2; r1 = r0*4

Multiplication w/o using Multiplier

- Example, multiply by 5

  ADD r0, r1, r1, LSL #2 ; r0 = r1 + r1*4

- Example, multiply by 7

  RSB r0, r2, r2, LSL #3 ; r0 = r2*8 –r2

  ; r0 = r2*7

- Hence, it is easy to multiply a number by a power of 2, a power of 2 ± 1 without using the multiplier.

Complex Multiplication

- More complex multiplication can be constructed from simpler multiplications, e.g. *35 can be made from *7 followed by *5.
- Example: multiply by 115

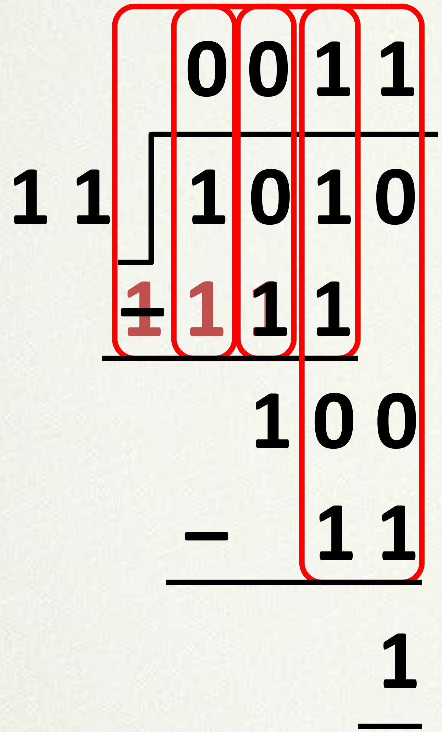
```
ADD r0, r1, r1, LSL #1 ; r0 = r1*3
SUB r0, r0, r1, LSL #4 ; r0 =(r1*3)-(r1*16)
                       ;    =r1*(-13)
ADD r0, r0, r1, LSL #7 ; r0 = (r1*-13) + (r1*128)
                       ;    = r1*115
```

Division

- ARM7 does not include a hardware binary integer divider mostly because division is so infrequently used. It can therefore can be done in software.
- A divider takes up to much silicon area and consumes too much power.
- N-bit division can be perform using N compare/subtracts
- Division is complicated!!!

Binary Integer Division

- Worked example: (4 bit unsigned: 10/3)



(quotient = 3)

(remainder = 1)

10 ÷ 3 = 3 remainder 1 ✓

Unsigned Division algorithm

- So to do the *n*-bit division of $D \div V$, the algorithm is;

1. set $i=n-1$, *set* $R=D$
2. compare $V<<i$ *with* R, *decrementing i until* $(V<<i) \leq R$
3. then set $R=R-(V<<i)$ *and put a 1 in the quotient (answer word* in bit position i) and repeat until $i=0$.
4. At the end, the quotient holds the answer, with remainder in R

- In previous example, n = 4 (for simplicity), D =10, V = 3

1. Set i = 3, R = D =10= 1010B (B for binary)
2. 11B << 3 = 11000B > 1010B (i=3)

   (Bit 3 in the quotient is 0)

   11B << 2 = 1100B > 1010B (i=2)

   (Bit 2 in the quotient is 0)

   11B << 1 = 110B < 1010B (i = 1)
3. Bit 1 in the quotient is set to 1 and R =1010B-110B = 100B
4. Repeat step 2, i = 0

   11B<100B, hence bit 0 in quotient is set to 1 and
   R=100B-11B=1B (Remainder), quotient = 11B

Signed Division

- It is difficult to perform division on the two’s complement number directly.
- Hence convert the signed number into unsigned number. Can use subtract instruction!!
- Perform unsigned division.
- Restore the sign of the number, again can use the subtract instruction.

Radix point and Q-format

- The IEEE floating point format is quite complicated and for many applications, programmers use a simpler Q-format.
- Qformat fractional numbers are often called "(x.y) format"

  (where x+y= total no. of bits in the word)
- To represent a fraction in fixed point binary arithmetic, just move the logical position of the radix point.
- There are signed and unsigned Q-format numbers. For example, here are some of the possible 16 bit signed Q format numbers:

Format	binary sequence	decimal range
Q0(16.0)	0000000000000000	-32768 to 32767
Q1(15.1)	000000000000000**0**	-16384 to 16383.5
Q2(14.2)	00000000000000**00**	-8192 to 8191.75
Q7(9.7)	000000000**0000000**	-256 to 255.9921875
Q15(1.15)	0**000000000000000**	-1 to 0.9999694824..

32-bit signed Q-format numbers

Format **binary sequence** **decimal range**

Q0(32.0) 0…0000000000000000 -2147483648 to 2147483647

Q1(31.1) 0…000000000000000**0** -1073741824 to 1073741823.5

Q2(30.2) 0…00000000000000**00** -536870912 to 536870911.75

…

Q15(17.15) 0..0**000000000000000** -65536 to 65536

...

Q31(1.31) 0**000000000000000…0** -1 to.9999999995

Conversion to Q-format

- Represent *e* (2.71828) in 16-bit precision Q13 format.
- Take *e* multiply by 2^{13} and then convert to hexadecimal.
- $e \times 2^{13} = 22{,}268.1647$
- Take whole part and convert to hex.
- 22,268 = 0x56FC = 0101011011111100

Sign bit

Radix point

Q-format Arithmetic

- The logical position of the radix point must be taken into account by the programmer (*this is not a hardware issue!!!*) for arithmetic operations:

- **8 bit Qformat addition/subtraction:**
- **The Qformat of both numbers must be the same !**

00000111 + **00000101** = **00001100**

Q2 (6.2) + Q2 (6.2) = Q2 (6.2) **1.75** + **1.25** = **3.00**

00000111 + **00000101** = **00001100**

Q2 (6.2) + Q3 (5.3) = Q???? **1.75** + **0.625** = **?.??**

8-bit Q-format multiplication

The Qformat of both numbers can differ, but must be known:

```
     6 5 4 3 2 1 2 1
     00000111
    11 10 9 8 7 5 4 3
×    00000101
-------------------
       111111
+      111
-------------------
0000000000100011
```

Q2 (6.2) × Q3 (5.3) = Q5 (11.5)

1.75 × 0.625 = 1.09375

What Qformat is the result? If we multiply numbers of format (n.m) and (p.q) then the resultant format is (n+p.m+q).

It means the multiply result is always twice as long as the operand length (which is normal for every binary multiplication).

Qformat arithmetic uses exactly the same hardware as normal arithmetic, that's why it's so useful!

32-bit Q-format Multiplication

- Example 1: multiply two 32-bit Q15 numbers
  - Q15 × Q15 or (17.15) × (17.15)
  - Of course the result is a (34.30) number
  - If we shift the result left by 17 bits the number becomes a (17.47) number. Drop the least significant 32 bits, we end up with a (17.15) number!
- Example 2: multiply a Q15 number with a Q0 number
  - Q0× Q15 or (32.0) × (17.15).
  - Of course the result is a (49.15) number.
  - Drop the most significant 32 bits, we end up with a (17.15) number
- Q31 has a maximum value of 1.0 and we know that if we multiply two numbers together that are both ≤ 1.0, then the result will also be ≤ 1.0... it means this format can guarantee no overflow when we do a multiply!

Summary

- Addition
- Subtraction
- Multiplication
- Division
- Q-format (fractional notation)