CS1021 Introduction to Computing I
2. ARM Assembly Language

Rebekah Clarke
clarker7@scss.tcd.ie

Registers


```
start
        MOV   total, a            ; Make the first number the subtotal
        ADD   total, total, b     ; Add the second number to the subtotal
        ADD   total, total, c     ; Add the third number to the subtotal
        ADD   total, total, d     ; Add the fourth number to the subtotal
```

Pseudocode program from Lecture #1

Adds four numbers together, total = a + b + c + d

total, a, b, c, and d are stored in memory

Operations (move and add) are performed in CPU

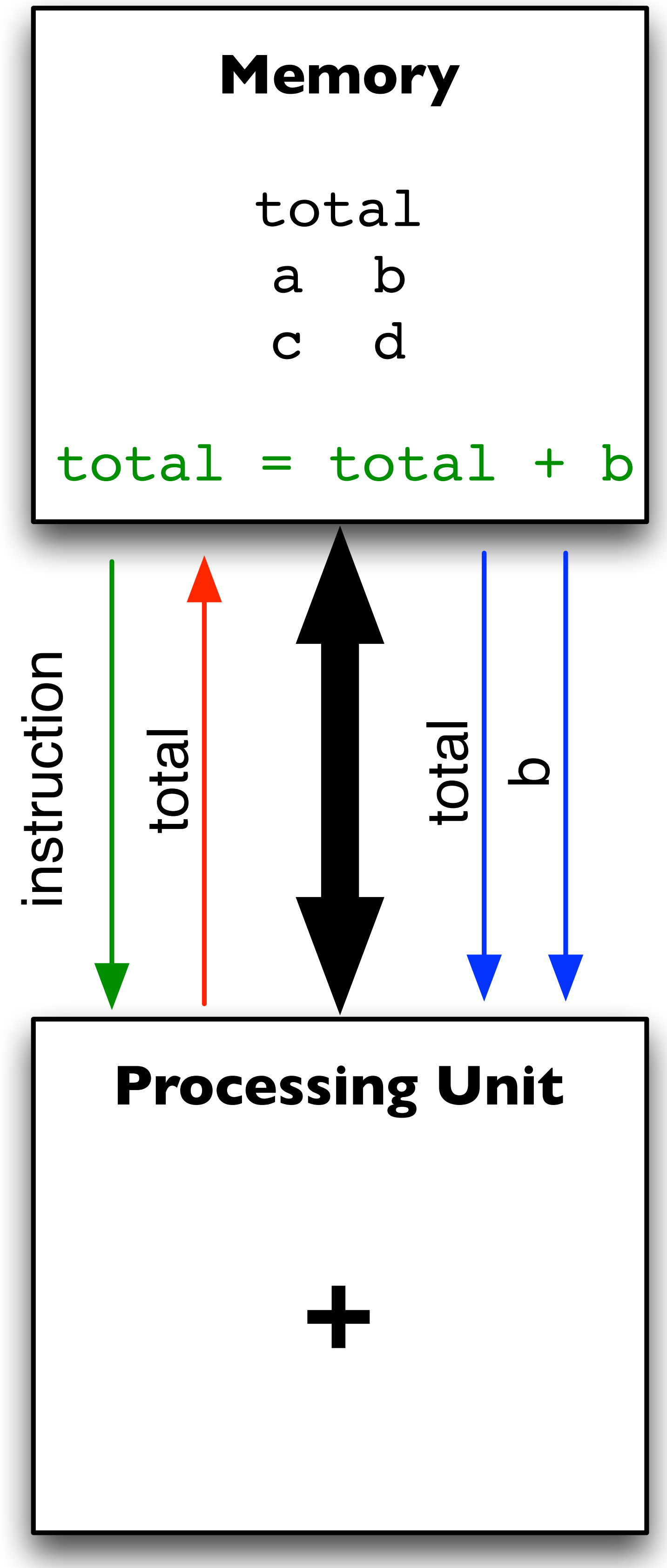
Several memory ↔ CPU transfers required

Registers

Accessing memory is slow relative to the speed at which the processor can execute instructions

total = total + b

- Load instruction from Memory to CPU (ADD total, total, b)
- Load total from Memory to CPU
- Load b from Memory to CPU
- Compute total + b
- Store total from CPU to Memory

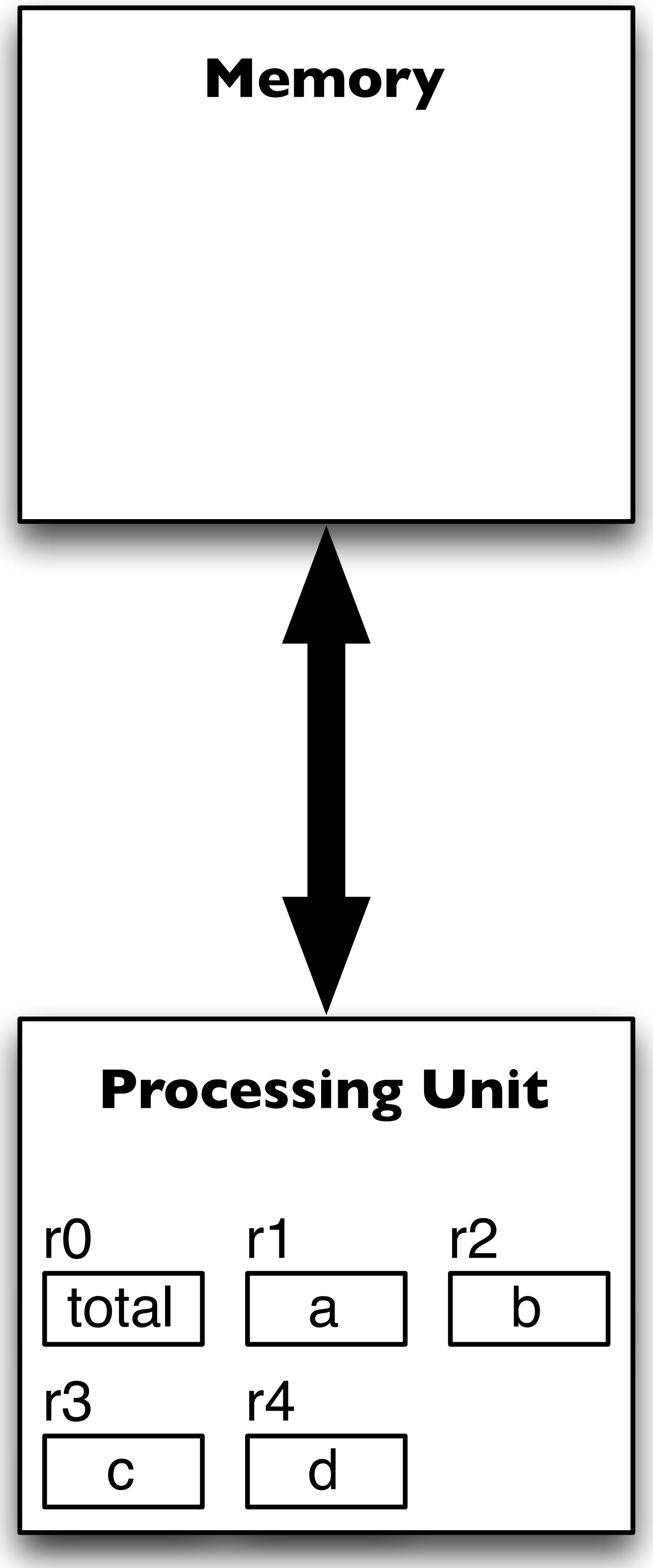


Registers

Processors use small fast internal storage to temporarily store values – called <u>registers</u>

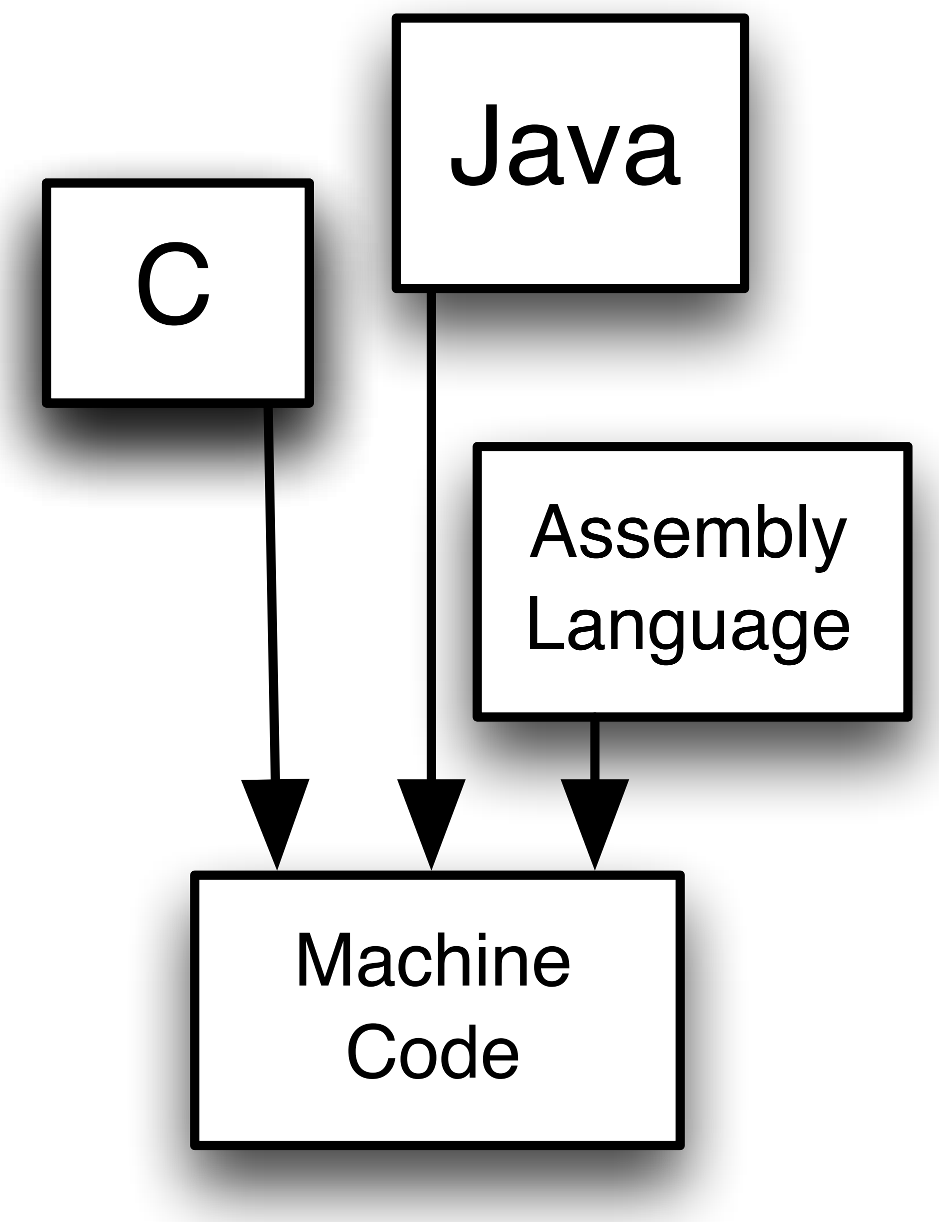
ARM has16 word-size registers

- Labelled r0, r1, ..., r15
- r15 is special – the Program Counter
- r13 and r14 are normally used for special purposes
- you should avoid using r13…r15 (for now …)



Machine Code

A program (any program, originally written using any language) is composed of a sequence of machine code instructions that are stored in memory



Instructions determine the operations performed by the processor (e.g. add, move, multiply, subtract, compare, ...)

A single instruction is composed of

- an operator (instruction)
- zero, one or more operands

Each instruction and its operands are encoded using a 32-bit value

- e.g. 0xE0810002 is the machine that causes the processor to add the values in r1 and r2 and store the result in r0

Assembly Language

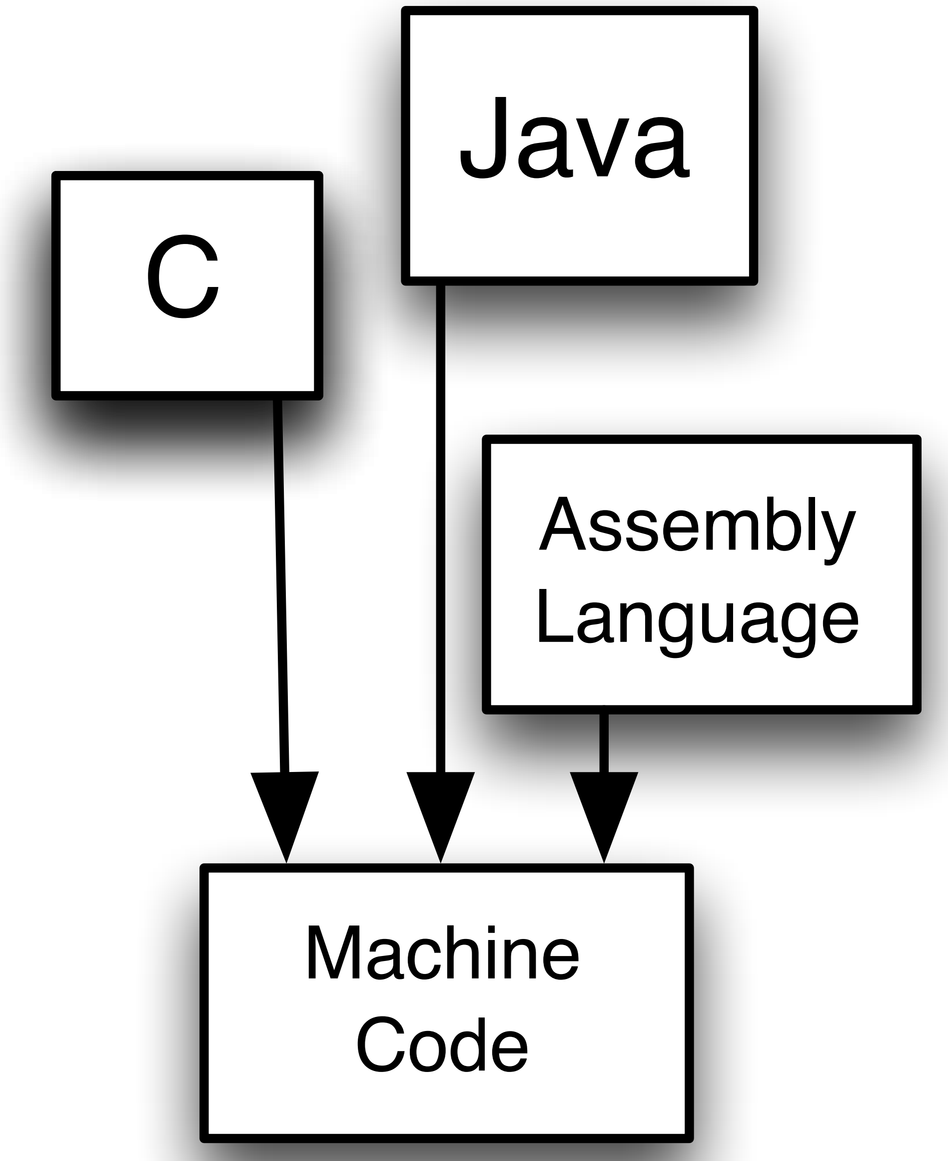
Writing programs using machine code is possible …
… but not practical

Instead, we write programs using assembly language

Instructions are expressed using mnemonics

e.g. the word “ADD” instead of the machine code 0xE08

e.g. the expression “r2” to refer to register number two



Assembly language must still be translated into machine code

Done using a program called an assembler

Machine code produced by the assembler is stored in memory and executed by the processor

Demo Program - ARM Assembly Language


```
start
        MOV   r0, r1              ; Make the first number the subtotal
        ADD   r0, r0, r2          ; Add the second number to the subtotal
        ADD   r0, r0, r3          ; Add the third number to the subtotal
        ADD   r0, r0, r4          ; Add the fourth number to the subtotal

stop    B     stop
```

Demo Program – Machine Code


```
 1 00000000                   AREA            Demo, CODE, READONLY
 2 00000000                   IMPORT          main
 3 00000000                   EXPORT          start
 4 00000000
 5 00000000          start
 6 00000000 E1A00001          MOV             r0, r1
 7 00000004 E0800002          ADD             r0, r0, r2
 8 00000008 E0800003          ADD             r0, r0, r3
 9 0000000C E0800004          ADD             r0, r0, r4
10 00000010
11 00000010 EAFFFFFE
                     stop     B               stop
12 00000014
13 00000014                   END
```

line #

instruction address

machine code instructions

original assembly language

Machine Code and Assembly Language

Every ARM machine code instruction is 32-bits long

32-bit instruction word must encode

- operation (instruction)
- all the required instruction operands

Example – `add r0, r0, r2`

Destination Operand
`Rd`

Source Operands
`Rn, Rm`

add instruction template
ADD Rd, Rn, Rm

1 1 1 0 0 0 0 0 1 0 0 0	Rn	Rd	0 0 0 0 0 0 0 0	Rm
31 … 20	19 … 16	15 … 12	11 … 4	3 … 0

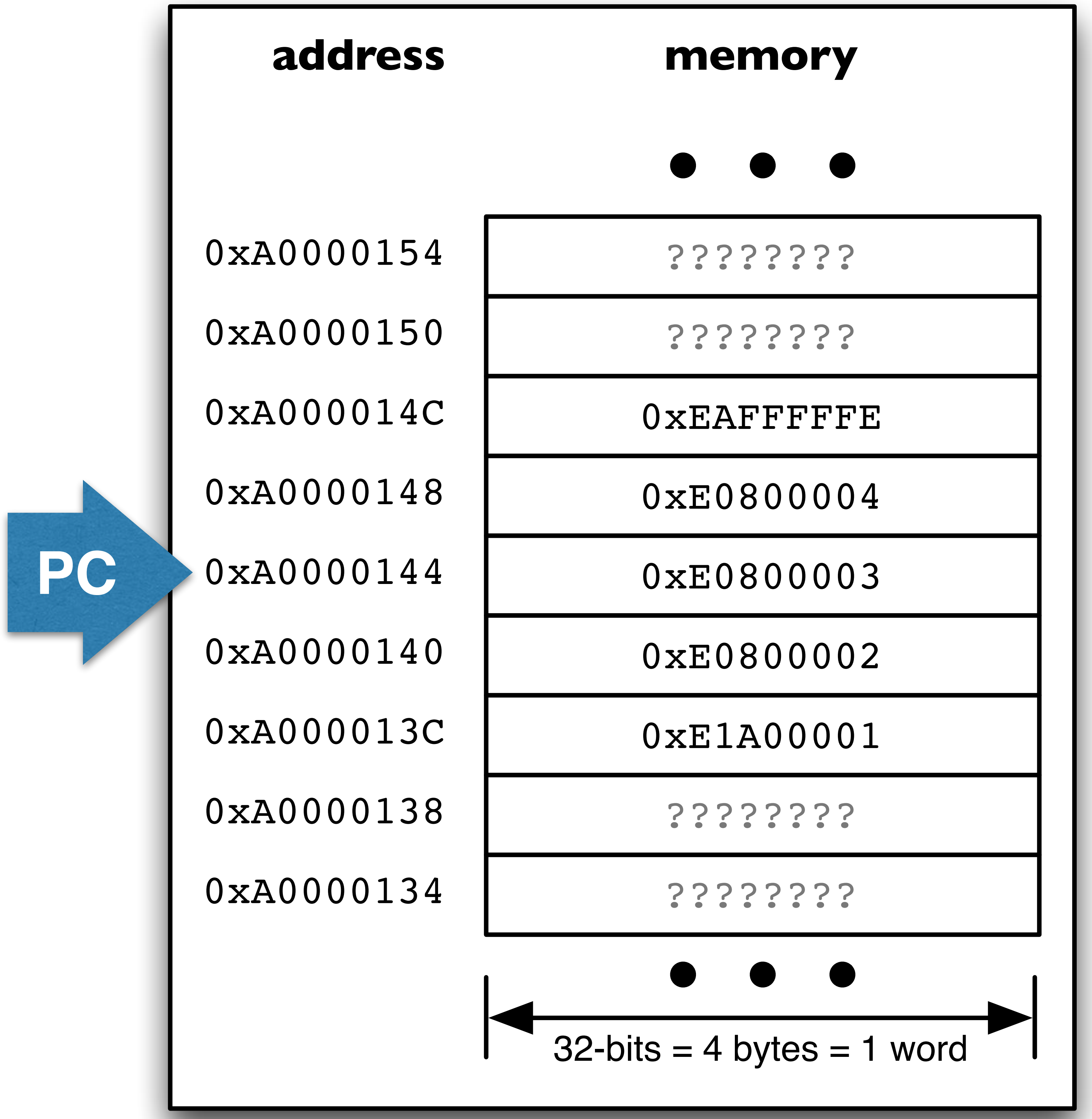
machine code binary

1 1 1 0 0 0 0 0 1 0 0 0	0 0 0 0	0 0 0 0	0 0 0 0 0 0 0 0	0 0 1 0
31 … 20	19 … 16	15 … 12	11 … 4	3 … 0

machine code hexadecimal

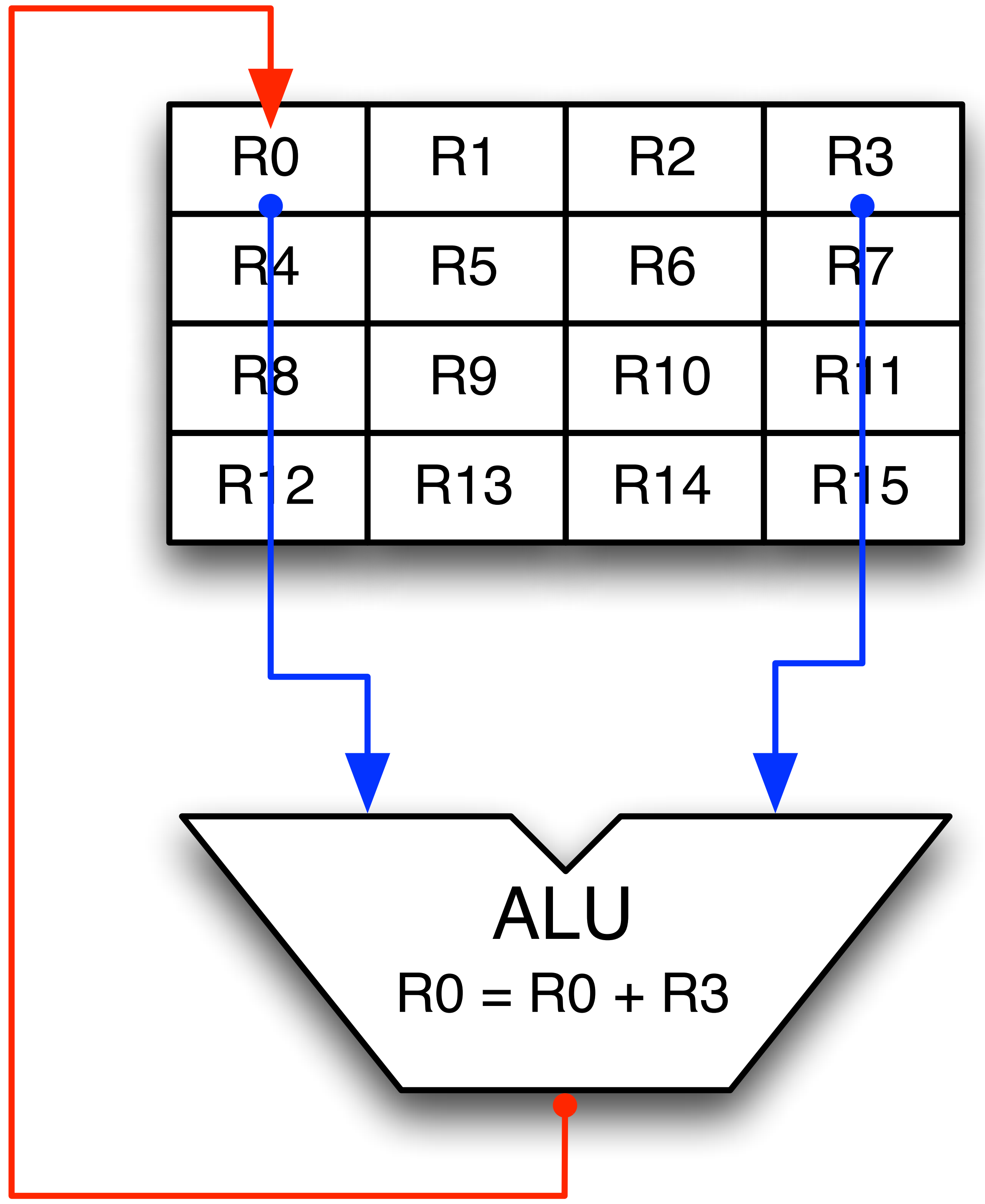
E	0	8	0	0	0	0	2
31 … 28	27 … 24	23 … 20	19 … 16	15 … 12	11 … 8	7 … 4	3 … 0

Instruction Execution



<u>**0xE0800003**</u>

operation:	add
source Rn:	R0
source Rm:	R3
destination:	R0



Fetch next instruction from memory at the address contained in the Program Counter (PC / R15)

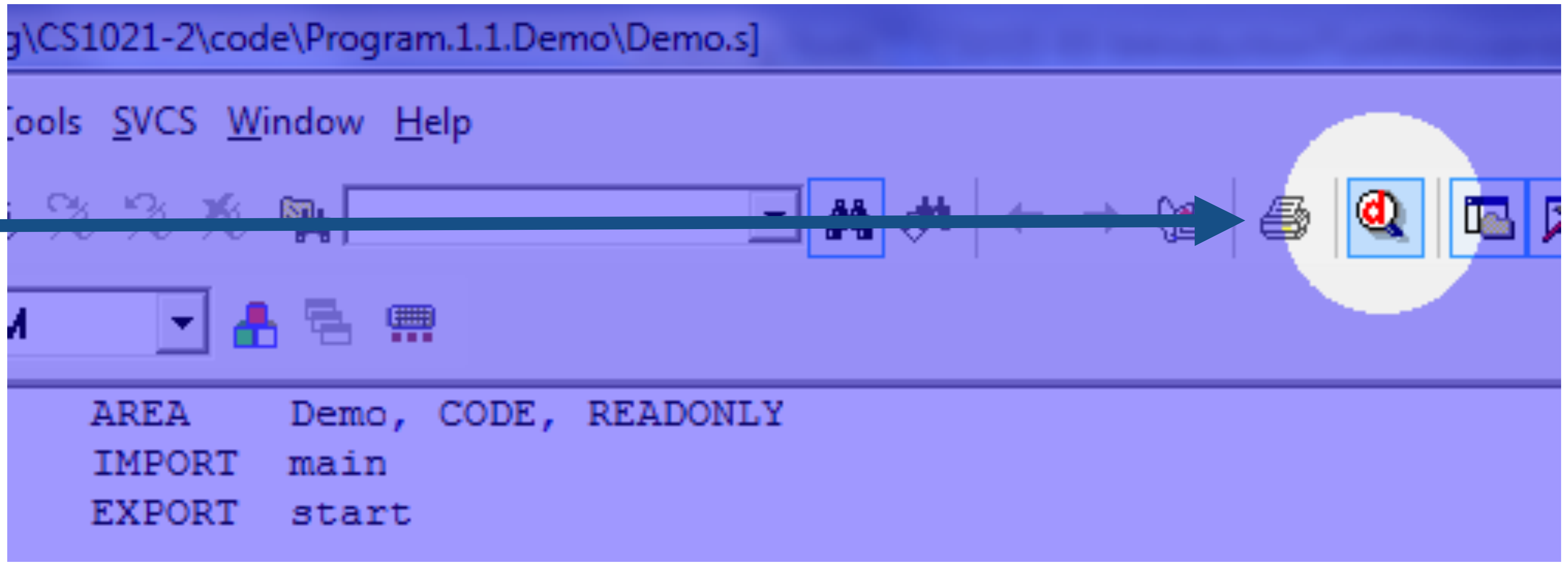
PC is advanced to next instruction (PC = PC + 4)

Machine Code instruction is **decoded** to determine operation and source / destination operands

Instruction is **executed**. (In this example the ALU adds the values in R0 and R3, storing the result back in R0.)

Program Execution (using uVision)

Start Debug session



Assembled program (Machine Code) loaded into memory as pre-defined address

Program Counter (PC / R15) set to same pre-defined address

Fetch-Decode-Cycle resumes

address	memory
	• • •
0xA0000154	????????
0xA0000150	????????
0xA000014C	0xEAFFFFFE
0xA0000148	0xE0800004
0xA0000144	0xE0800003
0xA0000140	0xE0800002
0xA000013C	0xE1A00001
0xA0000138	????????
0xA0000134	????????
	• • •

32-bits = 4 bytes = 1 word

What happens when we reach the end of our program?

Program: Swap Registers

Write a ARM Assembly Language program to swap the contents of registers R0 and R1

Immediate Operands

Register Operands

```
ADD   R0, R1, R2
MOV   R5, R2
```

Register Operand

Often want to use constant values, instead of registers

e.g. move the value 0 (zero) into register R3

```
MOV   R0, #0
```

Immediate Operands

e.g. set R1 = R2 + 1

```
ADD   R1, R2, #1
```

Simple Arithmetic

Write an ARM Assembly Language program to compute $4x^2+3x$ if x is stored in R1. Store the result in R0.

```
start
        MUL   r0, r1, r1            ; result = x * x
        LDR   r2, =4                ; tmp = 4
        MUL   r0, r2, r0            ; result = 4 * x * x

        LDR   r2, =3                ; tmp = 3
        MUL   r2, r1, r2            ; tmp = x * tmp

        ADD   r0, r0, r1            ; result = result + tmp

stop    B     stop
```

cannot use MUL to multiply by a constant value

MUL Rx, Rx, Ry produces unpredictable results

R1 is unmodified by our program … which may be something we want … or maybe we don't care

LoaD Register (LDR)


```
...
LDR   r2, =3              ; tmp = 3
MUL   r2, r1, r2          ; tmp = x * tmp
...
```

Note use of operand **=3**

Move constant value 3 into register R2

Loa**D** **R**egister instruction can be used to load any 32-bit signed constant value into a register

```
...
LDR   r4, =0xA000013C     ; r4 = 0xA000013C
...
```

Note use of **=x** syntax instead of **#x** with LDR instruction

MOV, LDR and Constant Values

Cannot fit large constant values in a 32-bit MOV instruction (Remember: all ARM instructions are 32-bit words)

```
        MOV   r0, #0x4FE8
```

```
error: A1510E: Immediate 0x00004FE8 cannot be represented by 0-255 and a rotation
```

LDR is a “pseudo-instruction” that simplifies the implementation of a word-around for this limitation

For small constant values, the Assembler quietly replaces the LDR instruction with a simple MOV instruction

```
        LDR   r0, =7
```

```
6 00000000 E3A00007          LDR                 r0, =7
```

```
        MOV   r0, #7
```

```
6 00000000 E3A00007          MOV                 r0, #7
```

Assembler transparently implements the work-around for us for large constant values

Assembly Language Programming Guidelines

Provide meaningful comments and assume someone else will be reading your code

```
MUL    r2, r1, r2                ; r2 = r1 * r2        ✗
```

```
MUL    r2, r1, r2                ; tmp = x * tmp       ✔
```

Break your programs into small pieces separated by white space

While starting out, keep programs simple

Pay attention to initial values in registers (and memory)