Lecture 2

Mnemonics

- In general nobody remembers all of the machine code for any particular processor (or indeed any).
- Instead we use mnemonics
  - mnemonics are words or phrases which are easy to remember and
  - can replace something which is difficult to remember.
- The instruction to move the value 114 into register r12 has the mnemonic
  - MOV r12, #114.
  - This is much easier to remember than 0xE3A0C072

Mnemonics

- Every instruction has both a machine code and a mnemonic.
- Consider the instructions from the last lecture.

Machine Code	Mnemonic
0xE3A0C072	MOV r12, #114
0xE3A070CB	MOV r7, #0xCB
0xE1A0600E	MOV r6, r14
0xE1A0700C	MOV r7, r12

Assembly Language

- If the mnemonics for every instruction in a computer program were listed in the order that they were executed then the resulting list would be an assembly language program.
- E.g.

```
MOV r6, r14
MOV r7, #0xCB
MOV r7, r12
MOV r12, #114
```

Assembler

- A computer package called an assembler converts an assembly language program into a machine code program.
- E.g.

```
0xE1A0600E
0xE3A070CB
0xE1A0700C
0xE3A0C072
```

- The machine code can be downloaded to the microprocessor memory;
- each instruction occupying 4 adjacent memory locations.

Compiler

- A high level language (such as Java, C++, Fortran, Pascal, etc.) is converted into either machine code or mnemonics using a computer package called a compiler.
- Most programs are written in a high level language
- but assembly language programming is commonly used for engineering systems which must operate in real time e.g. a mobile phone.
- Nobody writes computer programs using machine code.

Instructions for Arithmetic

- The ARM7 can add, subtract and multiply numbers (but not divide).
- The mnemonic for add the value in register x to the value in register y and place the sum in register z is: ADD rz, ry, rx
- E.g. to add the value in register r1 to the value in register r2 and leave the sum in register r3
  - the mnemonic is: ADD r3, r2, r1
  - We don’t need to know the machine code.

Subtraction

- Similarly the mnemonic for subtract the value in rx from the value in ry and place the difference in rz is: SUB rz, ry, rx
  - Note that the order is important and
  - if the value in rx is greater than the value in ry
  - then a negative result will be placed in rz.
  - We will return in negative numbers in another lecture.

Reverse Subtraction

- There is also a ‘reverse subtraction’ instruction;
- that is the same as subtraction but the registers are in the opposite order.
- The mnemonic is: RSB rz, ry, rx
- The meaning of the instruction is ‘subtract the value in ry from the value in rx and place the difference in rz’.

Multiplication

- The mnemonic for multiply the value in rx to the value in ry and place the product in rz is:

  MUL rz, ry, rx

- If the result is more than 32 bits long the destination register, rz, only holds the bottom 32 bits of the result and the rest is lost.

Overflow

- For example in decimal 5 000 000 000 is the product of 100 000 and 50 000.
- In hexadecimal: the product of 0x186A0 and 0xC350 is 0x12A05F200 which in binary is:
- $1\ 0010\ 1010\ 0000\ 0101\ 1111\ 0010\ 0000\ 0000_2$
- But the result is 33 bits long.
- The destination register holds the bottom 32 bits (0x2A05F200) and the most significant bit is lost.
- The same can happen with addition.

Multiply and accumulate

- Another very useful instruction is multiply and accumulate which has the following mnemonic:
- MLA rz, ry, rx, rw
- The instruction is ‘add the value in rw to the product of the value in rx and the value in ry and place the result in rz’.
- This instruction is extensively used by digital filters for signal processing; e.g. mobile phones.

Instructions using logic

- As well as arithmetic the ARM7 can also do logic such as AND and OR.
- AND value in rx with value in ry and leave the result in rz has the mnemonic:

```
AND rz, ry, rx
```

- OR value in rx with value in ry and leave the result in rz has the mnemonic:

```
ORR rz, ry, rx
```

Logical AND

- Logical functions are performed bit by bit so if we AND 0xFEDCBA98 with 0x11223344 :

```
     1111 1110 1101 1100 1011 1010 1001 1000
 AND 0001 0001 0010 0010 0011 0011 0100 0100
   = 0001 0000 0000 0000 0011 0010 0000 0000
```

- The result is 0x10003200.

Logical OR

- Similarly if we OR 0xFEDCBA98 with 0x11223344

```
      1111 1110 1101 1100 1011 1010 1001 1000
   OR 0001 0001 0010 0010 0011 0011 0100 0100
   ------------------------------------------
    = 1111 1111 1111 1110 1011 1011 1101 1100
```

- The result is 0xFFFEBBDC.

Exclusive OR

- Exclusive OR has mnemonic EOR rz, ry, rx
- To exclusive OR 0xFEDCBA98 with 0x11223344 :

```
        1111 1110 1101 1100 1011 1010 1001 1000
    EOR 0001 0001 0010 0010 0011 0011 0100 0100
      = 1110 1111 1111 1110 1000 1001 1101 1100
```

- The result is 0xEFFE89DC.

Bit clear

- ‘Bit clear’ performs the function ‘ry AND NOT(rx)’ and has mnemonic BIC rz, ry, rx
- To bit clear 0xFEDCBA98 with 0x11223344 :

```
      1111 1110 1101 1100 1011 1010 1001 1000
  BIC 0001 0001 0010 0010 0011 0011 0100 0100
    = 1110 1110 1101 1100 1000 1000 1001 1000
```

- The result is 0xEEDC8898.

Bit clear

- ‘Bit clear’ performs the function ‘ry AND NOT(rx)’ and has mnemonic BIC rz, ry, rx
- To bit clear 0xFEDCBA98 with 0x11223344 :

```
       1111 1110 1101 1100 1011 1010 1001 1000
   BIC 0001 0001 0010 0010 0011 0011 0100 0100
     = 1110 1110 1101 1100 1000 1000 1001 1000
       ----------------------------------------
```

- The result is 0xEEDC8898.

Addressing modes

- The concept of addressing mode is common to all microprocessors and an understanding of addressing mode is important for users of microprocessors.
- The ARM7 microprocessor supports many different addressing modes and we will concentrate on just four.

Register Addressing

- Register addressing means that the instruction code includes a number (or numbers) that identifies a register (or registers).
- E.g.
  - 0xE1A06002
    - which is the code for
    - MOV r6, r2
      - move into r6 the contents of r2.
- All of the instructions we have already covered use register addressing.
- E.g.
  - SUB r2, r1, r0
  - MUL r10, r11, r12
  - ORR r7, r6, r5

Immediate Addressing

- Immediate addressing means that the instruction code contains a value to be used.
- E.g.
  - 0xE3A0C0A0
    - which is the code for
    - MOV r12, #0xA0
      - move into r12 the value 0x000000A0 - the # means ‘immediate’.
- This instruction uses both register addressing for r12 and immediate addressing for 0xA0 (=160_{10})

More Immediate Addressing

- Immediate addressing can be used to replace the last register in any of the instructions we have covered so far (except MUL and MLA).
- E.g.
  - ADD r2, r1, #10
    - add 10 to value in r1 and
    - put the sum in r2
  - SUB r8, r8, #99
    - subtract 99 from value in r8
  - RSB r5, r14, #20
    - subtract value in r14 from 20 and
    - put the difference in r5
  - AND r12, r9, #0xFF
    - logical AND value in r9 with 0x000000FF and
    - put the result in r12

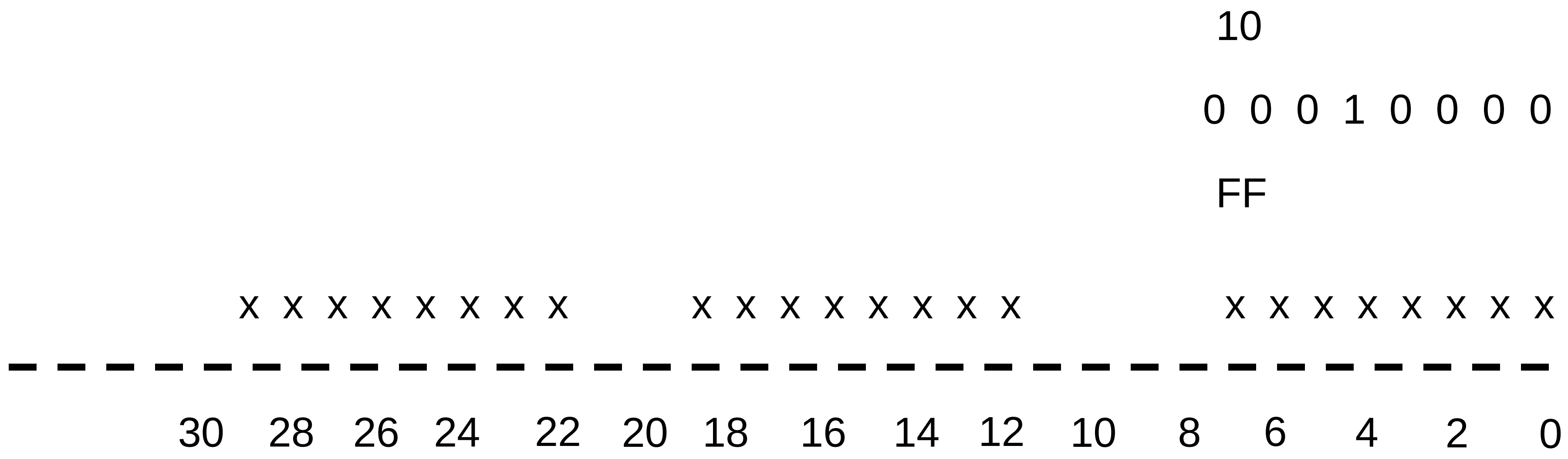
For Internal Use Only!

Problem!

- There is one problem with immediate addressing when using a processor like the ARM.
  - The instruction code is always 32 bits and
  - it must include information about the type of instruction
    - (e.g. ADD, MOV, EOR, etc.) and
    - the destination register as well as the immediate value.
  - So a 32 bit value can not be put into a 32 bit register
    - using immediate addressing and
    - MOV.
- E.g.
  - the following instruction is not allowed
  - MOV r1, #0xF97D5EC5 E1A01???

Restrictions

- The immediate value can only be one byte (8 bits) but it does not have to be the least significant byte
  - E.g.
    - MOV r4, #0xFF0 or
    - MOV r11, #0x3FC0000   (0011 1111 1100 …..)
  - are allowed and
  - these instructions would put
    - 0x00000FF0 into r4 and (0…0 1111 1111 0000)
    - 0x03FC0000 into r11.   (0011 1111   1100 0….0)
- The immediate value can be any value given by N $\times 2^{2M}$ where N is in the range 0 to 255 and M is in the range 0 to 15 . The instruction code for
- MOV rZ, #N $\times 2^{2(16-M)}$ is 0xE3A0ZMNN.

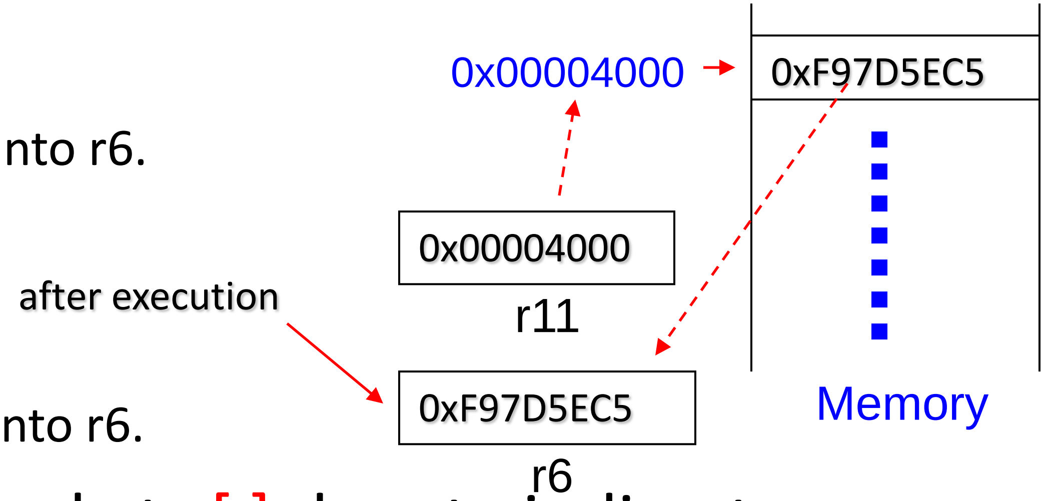

10
0 0 0 1 0 0 0 0
FF
x x x x x x x x
x x x x x x x x
x x x x x x x x
30 28 26 24 22 20 18 16 14 12 10 8 6 4 2 0

Indirect Addressing

- Indirect (or register indirect) addressing uses a value in a register to identify a memory address
  - e.g. the instruction:
    - LDR r6, [r11]
      - means put (or 'load') into register r6 the data held in the memory location that has the address given by the value in register r11.

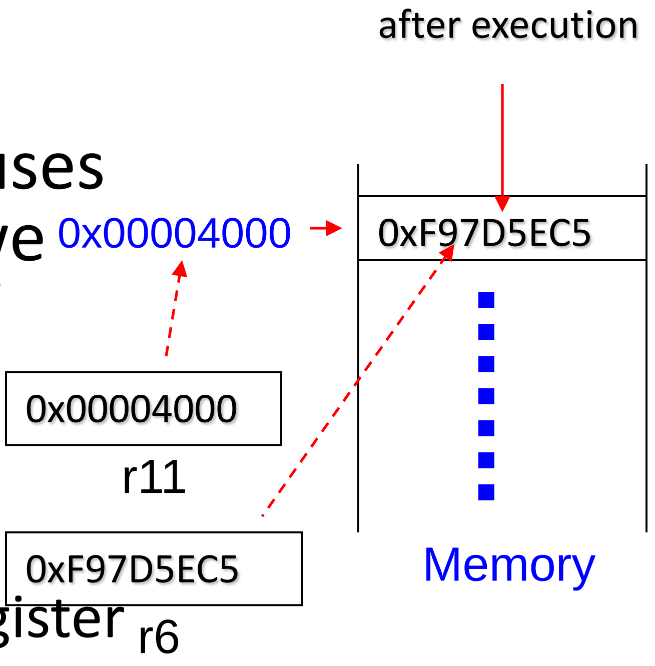
Indirect Addressing - Example

- So if r11 held the value 0x00004000 and the memory location with address 0x00004000 held the value 0xF97D5EC5 then:
  - LDR r6, [r11]
    - would load 0xF97D5EC5 into r6.
- Whereas
  - MOV r6, r11
    - would load 0x00004000 into r6.
- Note that the square brackets [ ] denote indirect addressing.



From register to memory.

- The load instruction, unlike MOV, can be used to put a true 32 bit value into a register.
- Another important instruction that uses indirect addressing is 'store' which we can think of as being the opposite of 'load'.
  - The instruction
  - STR r6, [r11]
  - means put (or 'store') the data from register r6 into the memory location that has the address given by the value in r11.



Load and Store

- When a load instruction is executed
  - the data travels from memory to register
- whereas when a store instruction is executed
  - the data travels from register to memory.
- Each memory location holds one byte or 8 bits of data whereas each register holds 4 bytes of data.
- So LDR and STR use four consecutive memory addresses but which byte goes to which location?

Little endian

- Microprocessors can be either ‘little endian’ or ‘big endian’
- If the processor is ‘little endian’ then the instruction:
  - STR r6, [r11]
  - with 0xFFAABB11 in r6 and 0x00008000 in r11
  - would store
    - byte 0x11 at address 0x00008000
    - 0xBB at 0x00008001
    - 0xAA at 0x00008002
    - 0xFF at 0x00008003

Big endian

- Whereas if the processor is ‘big endian’ then the instruction:
  - STR r6, [r11]
  - with 0xFFAABB11 in r6 and 0x00008000 in r11
  - would store
    - byte 0xFF at address 0x00008000
    - 0xAA at 0x00008001
    - 0xBB at 0x00008002
    - 0x11 at 0x00008003

Little endian or Big endian

- For little endian, the least significant byte (‘the little end’) is stored at the lowest memory address
- whereas for big endian, the most significant byte (‘the big end’) is stored at the lowest address.
- The ARM processor can be configured as either little endian or big endian.
- Intel (e.g. the Pentium) uses little endian whereas Motorola uses big endian.

Half words and bytes

- Load and store can also use half words and bytes.
- The instructions LDRH and LDRB load the lowest 16 or 8 bits respectively of a register from a memory location given by indirect addressing.
- The remaining 16 (for LDRH) or 24 bits (for LDRB) of the register are set to zero.
- Similarly STRH and STRB store either the lowest 16 or 8 bits of a register at a memory location given by indirect addressing.

Base plus offset addressing

- Base plus offset addressing uses a value in a register (the ‘base’) plus a binary number (the ‘offset’) to identify a memory address.
- E.g.
  - LDR r6, [r11, #12]
    - means load into r6 the data held in the memory location that has the address given by the value in register r11 added to 12.

Automatic updating

- In many applications there is a great deal of data movement between the CPU and memory and it can be very useful if the base register is updated on each load or store.
- The instruction:
  - LDR r6, [r11, #12]**!**
  - does the same as the instruction on the previous slide
  - but 12 is added to the value in r11.
  - The automatic updating is identified by the **!**, ‘pling’.

Pre-indexed and post-indexed

- The instruction:
  - LDR r6, [r11, #12]!
  - is an example of pre-indexing
    - meaning that 12 to added to the base register, r11, before it is used as a memory address.
- The instruction:
  - LDR r6, [r11], #12
  - is an example of post-indexing
    - the 12 is added to the base register, r11, after it is used for the memory address.
- There is no pling, !, for post-indexing because the base register is always updated.

Instruction set design

Fundamentally a CPU is designed to implement a particular ‘instruction set’, meaning all of the possible machine code instructions that can be executed on a particular CPU.

What should be included in the instruction set?

Computer programmers use high level languages such as C or Java which do not match machine code instructions.

> For each high level program instruction (e.g. printf) there will be many machine code instructions required to perform the same function.

Instruction set - complexity

A special (usually very complex) computer program called a compiler, translates each high level language instruction into many machine code instructions.

The difference between machine code and high level languages is called the semantic gap.

> If the CPU is very complicated (CISC) then it can perform more complicated instructions, thereby reducing the semantic gap so that the compiler is simpler.
>
> If the CPU is very simple (RISC) then the semantic gap is larger and the compiler is becomes very complicated.

CISC v. RISC

CISC = Complex Instruction Set Computer

1. Complicated CPU
2. Each instruction takes longer to execute
3. Fewer machine code instructions for each high level instruction
4. Good code density
5. Smaller semantic gap
6. Simple compiler

RISC = Reduced Instruction Set Computer

1. Simple CPU
2. Machine code instructions execute quickly
3. More machine code instructions for each high level instruction
4. Poor code density
5. Larger semantic gap
6. Complicated compiler

Code density?

Code density is a measure of the size of a computer program in memory for a given function.

Good code density produces smaller programs leading to lower memory cost and less power dissipation in memory.

There are a number of factors effecting code density.

1) How many bits in each machine code instruction.
2) The functionality of individual machine code instruction.
3) How good the compiler is.

Instruction set - considerations

What kinds of instructions are needed?

Data movement instructions – including to/from memory?

What ALU instructions are needed?

    Addition,
    subtraction,
    multiplication,
    division?

Should ALU instructions take values from memory and return results to memory?

Program control instructions, branches or jumps?

Support for more advanced memory data structures such as stacks?

4-operand (address) instruction format

f bits	n bits	n bits	n bits	n bits
function	op 1 addr.	op 2 addr.	dest. addr.	next i addr.

The most general form of a machine code instruction has four operands and some bits to identify it's function.

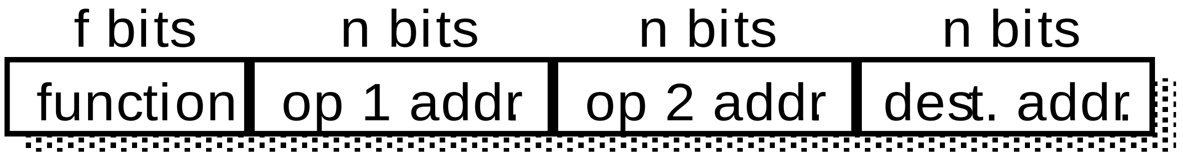
E.g. the f bits could signify addition.

The instruction would add 'operand 1' to 'operand 2' and place the result (sum) in a given destination.

The two operands and the destination could be given by either memory addresses and a number identifying an internal CPU register.

The final operand would identify the memory address of the next instruction to be performed.

3-operand instruction format



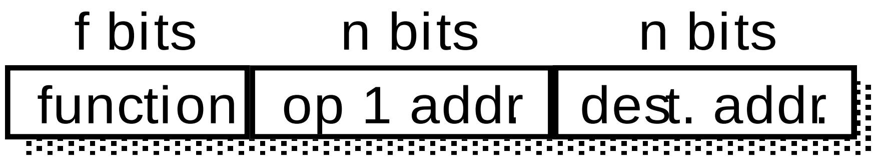
The first way to reduce the size of the instruction from (f+4n) bits to (f+3n) bits is to make the address of the next instruction implicit.

E.g. for a 32 bit processor (like the ARM) the next instruction is always 4 bytes after the current instruction except….

Instructions that modify the program sequence, such as ‘branch’ or ‘jump’, must specify the address of the next instruction.

An example of a 3 operand instruction is ADD in the ARM instruction set where three internal CPU registers are specified for the source (x2) and result.

2-operand instruction format



The size of the instruction can be reduced to (f + 2n) bits by using the destination operand as one of the source operands.

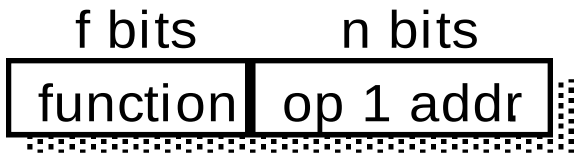
For an ADD instruction, the implemented function would be

d := d + s1;

where d is the value held by the destination and s1 is the value held by the source.

Two operand instructions are implemented by the Thumb instruction set (with operands held in internal CPU registers, not memory).

1-operand (accumulator) instruction



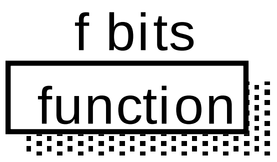
A further reduction to (f + n) bits can be achieved by using an implicit destination register, often called the ‘accumulator’.

For an ADD instruction, the implemented function would be

accumulator := accumulator + s1;

The MU0 processor described in Furber, section 1.3, is an example of a one operand instruction architecture.

0-operand instruction format



The simplest form of instruction architecture is achieved by making all operand references implicit; often using the stack.

E.g. For an ADD instruction, the function would be

top_of_stack := top_of_stack + next_on_stack;

The transputer designed by Inmos in the 1980's uses this architecture.

The **transputer** was a pioneering microprocessor architecture of the 1980s, featuring integrated memory and serial communication links, intended for parallel computing. It was designed and produced by Inmos, a semiconductor company based in Bristol, United Kingdom -- http://en.wikipedia.org/wiki/Transputer