

# Lecture 1

---

CPU

Microprocessor

## ARM7TDMI microprocessor

---

- T: 16-bit **T**humb code
- D: on-chip **D**ebug support
- M: an enhanced **M**ultiplier
- I: Embedded **I**CE hardware to give on-chip breakpoint and watchpoint support

## Revision - binary, hexadecimal,

---

0x denotes a hexadecimal number

## Representing characters

---

### ASCII

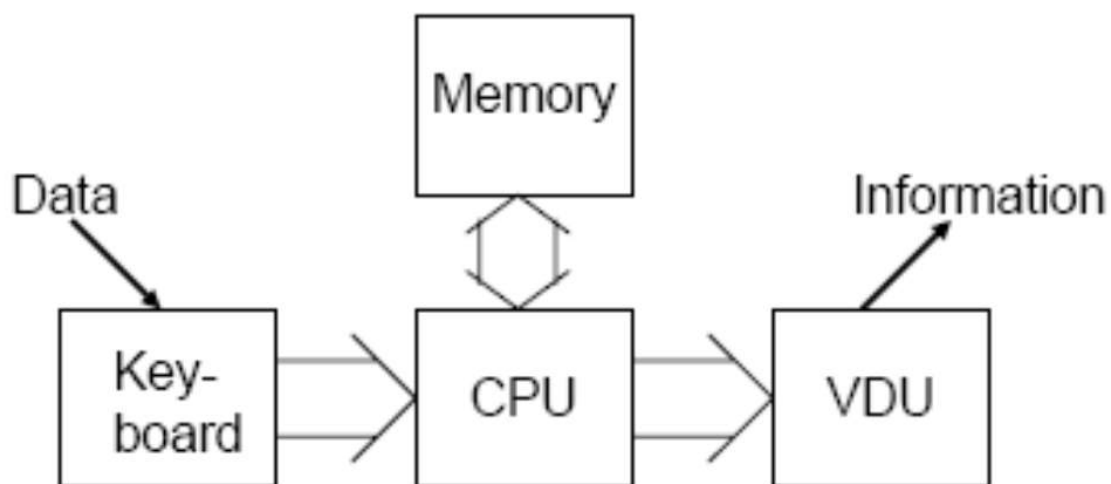
American Standard Code for Information Interchange

7 bit code

- the standard 26 characters of English in both upper and lower case
- characters such as !#\$%&()\*+,-./:;<=>?@[\\]^\_`
- the characters for numbers 0 to 9
- control 'characters' such as line feed, carriage return, delete, escape, backspace.
- It does not code for é or α or thousands of other non-English characters.

## A simple computer architecture

---



Data and information pass between the blocks as electrical signals.

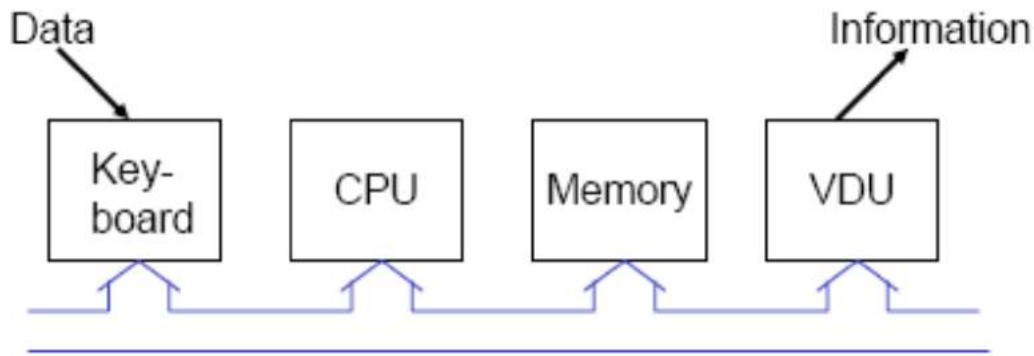
However this architecture can become very complicated if

- more than one input device or
- more than one output device needs to be connected

## The Bus Architecture

---

The bus architecture can be extended over any number of devices. All devices have only one connection onto the 'bus'.



A bus is a collection of electrical connections, normally 8, 16, 32 or 64 individual wires.

## Controlling the Bus

It is important that signals do not collide on the bus - only one device at a time can send data.

The **CPU controls** all movements on the bus using special wires to activate devices and to synchronize the sending and receiving devices.

These special connections are known as the **control bus** and they are completely **independent** of the bus along which data (called **data bus**) is passed.

## A Third Bus: Address Bus

The address bus is used by the CPU to determine which location in memory is sending or receiving data.

## What is in memory?

---

Memory is used to

- store the instructions which the CPU uses to process the data.
- store data in the form of numbers or characters.

Computer memory is **a very big sequential logic circuit made up of thousands or millions of simple logic gates**, such as a D type latch, which can remember a 0 or a 1, that is one bit of data.

Groups of these gates are collected together in a **memory 'location'**. There are typically 8 bits of data (aka, **byte**) in one location.

**Each memory location has a unique memory address.**

## Memory organization

Example: ARM7TDMI microprocessor

- At each location, it has 8 bits (1 byte) of data.

- The ARM is a 32 bit processor and addresses are 32 bits long from `0x00000000` to `0xFFFFFFFF`.
- In theory, up to 4,294,967,296 (or  $2^{32}$ ) different memory locations.
- In practice not all addresses are used for memory.

## Some definitions

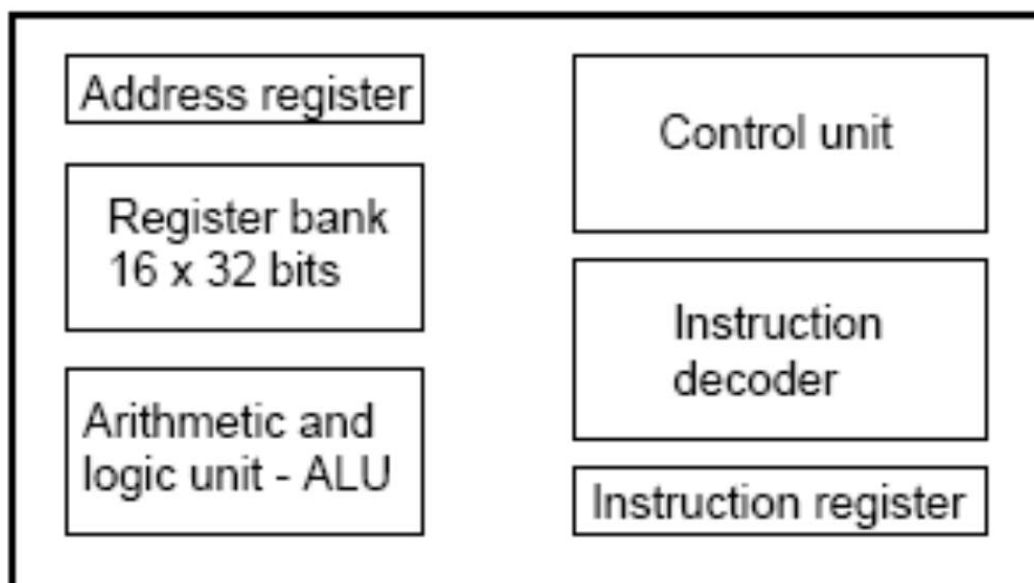
- 1 byte = 8 bits
- 1 kilobyte = 1024 bytes
- 1 megabyte = 1024 kilobytes
- 1 gigabyte = 1024 megabytes
- A 32 bit processor could be directly connected to 4GB of memory using a 32 bit address bus if every memory address had memory connected.
- `word` depends upon the processor used.
  - for a 32 bit processor like the ARM
  - 1 word = 32 bits = 4 bytes
  - word length = 32

## CPU

---

- interprets the instructions stored in memory.
- performs the calculations.
- controls the flow of data along the data bus.
- determines which memory address to use.

## The ARM7 core



## Instruction register

- The instructions stored in memory travel along the data bus to the CPU where they are loaded into the instruction register.
- ARM7 instructions are 32 bits long so the instruction register is a **32 bit memory device** - not part of the main memory.

- **fetch**: the process of loading the instruction register from memory.

## Instruction decoder, Control unit

The instructions are in 'machine code' and the instruction decoder **determines the function of each instruction**.

The instruction decoder and control unit **determine what the other parts of the CPU do**.

The control unit is also **in charge of the control bus**.

**decode cycle**: the process of interpreting each instruction.

## Arithmetic and Logic unit

The arithmetic and logic unit or ALU performs the mathematical functions as required.

**execute cycle**: the process of performing each instruction

## Register bank

A local memory for the CPU.

It has **16** locations - each location can hold 32 bits of data.

The registers are named **r0**, **r1**, **r2**, **r3**, ... etc. up to **r15**.

## Address register

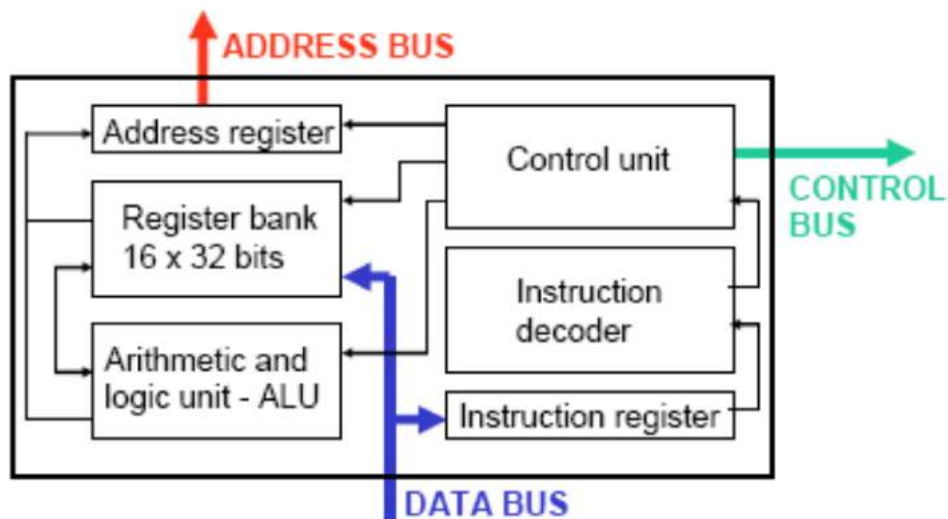
A 32 bit memory device which holds a memory address value.

- Either this address may be for the memory location of the next instruction during the **fetch** cycle.
- Or during the **execute** cycle the address is for a memory location either containing data to be loaded into a register or where data from a register is to be stored.

## Fetch, decode, execute.

- During the fetch cycle an instruction in memory is loaded into the instruction register.
- During the decode cycle the instruction is interpreted by the instruction decoder.
- During the execute cycle
  - either the ALU performs a calculation on values held in registers or
  - a value in a register is stored into memory or • a value in memory is loaded into a register.

## Internal & External connections



## Program counter

Register `r15` always holds the memory address of the next instruction to be executed.

## Instructions stored in memory

One instruction (32 bits long) would be stored in 4 memory locations

When an instruction is executing, the program counter, `r15`, increments by 4 so that it holds the memory address of the next instruction.

EXCEPT when a 'branch' instruction is executed when the computer program 'branches' to another part of memory and the program counter holds a completely new memory address.

## Machine code

`0xE3A0C072`

- `E3A0`
- `C`
- `72`

Move 114 into register `r12`

`0xE1A0600E`

- `E1A0`
- `6`
- `E`

Move into `r6` the value held in `r14`



# Lecture 2

---

## Mnemonics

---

words or phrases which are easy to remember.

`MOV r12, #114` instead of `0xE3A0C072`

## Assembly Language

---

### Assembler

converts an assembly language program into a machine code program.

### 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.

Nobody writes computer programs using machine code.

## Instructions for Arithmetic

---

The ARM7 can add, subtract and multiply numbers (but not divide).

### Addition

`ADD rz, ry, rx`

add the value in `register x` to the value in `register y` and place the sum in `register z`.

### Subtraction

`SUB rz, ry, rx`

subtract the value in `rx` from the value in `ry` and place the difference in `rz`

Note that the order is important.

### Reverse Subtraction

`RSB rz, ry, rx`

subtract the value in `ry` from the value in `rx` and place the difference in `rz`

## Multiplication

`MUL rz, ry, rx`

multiply the value in `rx` to the value in `ry` and place the product in `rz`

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. The same can happen with addition.

## Multiply and accumulate

```
MLA rz, ry, rx, rw
```

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

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

AND value in `rx` with value in `ry` and leave the result in `rz`

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

OR value in `rx` with value in `ry` and leave the result in `rz`

## Exclusive OR

```
EOR rz, ry, rx
```

## Bit clear

```
BIC rz, ry, rx
```

performs the function `ry AND NOT(rx)`

## Addressing modes

---

### Register Addressing

the instruction code includes a number (or numbers) that identifies a register (or registers).

### Immediate Addressing

the instruction code contains a value to be used.

```
0xE3A0C0A0 MOV r12, #0xA0
```

move into r12 the value 0x000000A0

`#` means 'immediate'.

### More Immediate Addressing

- `ADD`, `SUB`, `RSB`, `AND`

## Problem

a 32 bit value can not be put into a 32 bit register

## Restrictions

The immediate value can only be one byte (8 bits) but it does not have to be the least significant byte.

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.



```
MOV rZ, # $N \times 2^{\{2(16-M)\}}$
```

```
0xE3A0ZMNN
```

## Indirect Addressing

uses a value in a register to identify a memory address

### LOAD

```
LDR r6, [r11]
```

put (or 'load') into register `r6` the data held in the memory location that has the address given by the value in register `r11`.

square brackets `[ ]` denote indirect addressing.

The load instruction, unlike `MOV`, can be used to put a true 32 bit value into a register.

### STORE

```
STR r6, [r11]
```

put (or 'store') the data from register `r6` into the memory location that has the address given by the value in `r11`.

## Endian

Each memory location holds 1 byte of data whereas each register holds 4 bytes of data.

### Little endian

the least significant byte ('the little end') is stored at the lowest memory address

- `STR r6, [r11]`
- with `0xFFAABB11` in `r6` and `0x00008000` in `r11`
- would store byte
  - `0x11` at `0x00008000`
  - `0xBB` at `0x00008001`
  - `0xAA` at `0x00008002`
  - `0xFF` at `0x00008003`

### Big endian

the most significant byte ('the big end') is stored at the lowest address.

- `STR r6, [r11]`
- with `0xFFAABB11` in `r6` and `0x00008000` in `r11`
- would store byte
  - `0xFF` at `0x00008000`
  - `0xAA` at `0x00008001`
  - `0xBB` at `0x00008002`
  - `0x11` at `0x00008003`

**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.

Load and store can also use half words and bytes.

## 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

uses a value in a register (the 'base') plus a binary number (the 'offset') to identify a memory address.

```
LDR r6, [r11, #12]
```

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

```
LDR r6, [r11, #12]!
```

does the same as `LDR r6, [r11, #12]`, but `12` is added to the value in `r11`.

! 'pling'

# !!! Important

## Pre-indexed and post-indexed

```
LDR r6, [r11, #12]!
```

is an example of **pre-indexing**, meaning that `12` is added to the base register, `r11`, before it is used as a memory address.

```
LDR r6, [r11], #12
```

is an example of **post-indexing**, meaning that `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

### Complexity

### Semantic gap

The difference between machine code and high level languages.

## 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

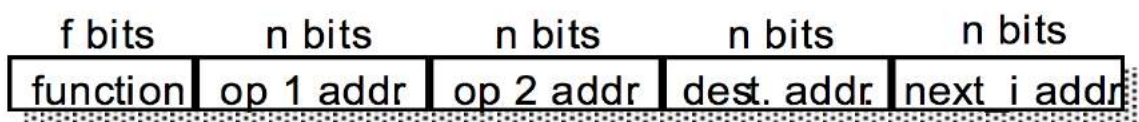
a measure of the size of a computer program in memory for a given function.

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.

## 4-operand (address) instruction format

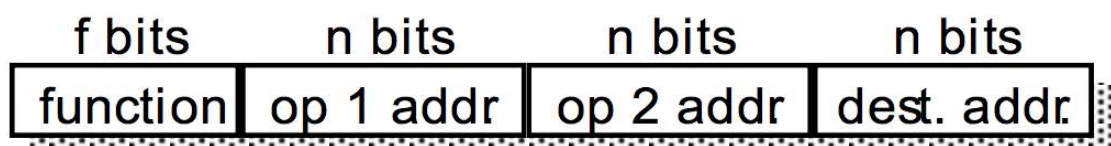


The most general form of a machine code instruction

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

The last 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.

## 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.



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'.

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.

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.

...

# Lecture 3

---

## Flags

---

also known as `condition codes`.

Flags are used to give a signal that something either has or has not happened.

Flags are essentially **1 bit memory devices** and different microprocessors have different flags.

- the zero flag `Z`
- the negative flag `N`
- the carry flag `C`
- the overflow flag `V`

An ARM instruction that would set the zero flag is :

```
SUBS r2, r1, r1
```

Note that the mnemonic is `SUBS` rather than `SUB`.

They perform the same function except `SUBS` will set or clear the flags and `SUB` will not change the flags from their current setting.

## Instructions

The following instructions do not alter the flags:

```
MOV, ADD, SUB, RSB, MUL, MLA, AND, EOR, ORR, BIC
```

The following instructions do **set or clear the flags** as required:

```
MOVS, ADDS, SUBS, RSBS, MULS, MLAS, ANDS, EORS, ORRS, BICS
```

## Usage

Flags are used in two ways:

1. Main usage: **conditional execution**, determine if another instruction is executed or not.
2. Used in some arithmetic instructions as **an additional value**.

## Conditional Execution

an instruction either does or does not execute depending upon the state of the flags (or 'condition codes').

Almost all ARM instructions can be conditionally executed and this is **indicated by the addition of two letters** in the mnemonic of the instruction.

## Condition Fields

There are **15** different condition fields which may be appended to (almost) any mnemonic.

Most common ones:

- `EQ` - 'equal' - executed only if the `Z` is set.

- **NE** - 'not equal' - executed if **Z** is clear.
- **CS** - 'carry set' - executed if **C** is set.
- **CC** - 'carry clear' - executed if **C** is clear.
- **MI** - 'minus' - executed only if N is set.
- **PL** - 'plus' or 'positive' - executed if N is clear.
- **VS** - 'overflow' - executed if V is set.
- **VC** - 'no overflow' - executed if V is clear.
- There are 7 more condition fields including 'always' (**AL**) which is the default if no condition field is specified.

## Branches

---

### Unconditional Branches

An unconditional branch always executes and it reloads the program counter with a new value.

The purpose of a branch instruction is to continue the program at a different location in memory.

### Conditional Branches

Unconditional branches are of very little use but conditional branches have many uses.

**BNE 0x08C00000**

branch to 0x08C00000 if not equal.

### Restrictions

The memory address in the branch instruction must be contained within the instruction itself.

**24 bits of the 32 bit** instruction code are used to determine the destination address of the branch as an offset from the current program counter value.

This means that the destination address must be within **±32MB** of the memory address of the branch instruction.

### Mnemonics

In general assembly language programs are written without any knowledge of the memory address for the corresponding machine code.

So the mnemonic for branch uses **labels** rather than actual memory addresses and the assembler program determines the actual value used.

### Use of the zero flag

The zero flag is commonly used to **test if a program 'loop' has been executed a certain number of times**;

### The carry flag

The carry flag is used in many arithmetic and logic instructions (best illustrated by addition).

- In common with the other flags it can be used to determine if a conditional instruction is executed or not.

- The other use of the carry flag is in the addition instruction `ADC` (and the subtraction instructions `SBC` and `RSC`).

## ADC

The instruction ADC 'add with carry' adds together the two values and adds another 1 if the carry flag is set.

`ADC` is used when we add together numbers greater than  $(2^{32}-1)$ .

- `r0` holds `0xCB417800` and `r1` holds `0x00000002`
- `r2` holds `0x42770C00` and `r3` holds `0x00000003`

```
ADDS r4, r2, r0
```

The carry flag is set and `r4` holds the lowest 32 bits: `0x0DB88400`.

```
ADC r5, r3, r1
```

Because the carry flag is set, an extra 1 is added into the sum so `r5` will hold `0x00000006`.

Taken together `r5` and `r4` hold the value `0x60DB88400` which is \$26,000,000,000\_{10}

## Negative numbers

- 1) Sign magnitude.
- 2) Two's complement.

### Sign bit

In each case the most significant bit - **m.s.b.** - indicates the sign (1 for -ve and 0 for +ve).

To find out if a number is -ve or +ve

```
MOVS rx, rx
```

The value in register `rx` remains unchanged but the negative flag is set if the **m.s.b.** is `1` and it is cleared if the **m.s.b.** is `0`.

### Sign magnitude

a negative number is the same as a positive number but with the **m.s.b.** equal to `1`.

In general any hexadecimal number is negative if it starts with 8 or greater and it is positive if it starts with 7 or less.

The magnitude of a 'sign magnitude' number is easy to find: simply `AND` with `0x7FFFFFFF`.

However **sign magnitude numbers cannot be used for arithmetic.**

### 2's complement

a negative number, `-x`, is given by the value `2n - x` for an n bit processor.

```
-20640
```

1. Find the positive value: `0x000050A0`
2. Invert all bits (`0 -> 1`, `1 -> 0`): `0xFFFFAF5F`
3. Add `1`: `0xFFFFAF60`

Unlike sign magnitude, arithmetic is simple in two's complement.

In two's complement any hexadecimal number which has an 8 or greater for the most significant digit is negative since the **m.s.b.** is 1.

## Subtraction

such as  $x - y$ , is implemented in a microprocessor by finding the two's complement of  $-y$  and then performing the addition,  $x + (-y)$

## The overflow flag **V**

is set because there is an overflow into the **m.s.b.**.

## Flags - summary.

---

- The zero flag, **Z**, is set when the result (not including any carry out) is **0x00000000**.
- The negative flag, **N**, is set when the most significant bit of the 32 bit result is 1.
- The carry flag, **C**, is set when the result (taken as an unsigned integer) is greater than  $(2^{32} - 1)$ .
- The overflow flag, **V**, is set when the result (taken as a two's complement number) is greater than  $(2^{31} - 1)$  or less than  $-2^{31}$ .

## Floating point numbers

---

$A \times 2^n$

Using the IEEE 754 standard the **A** value has a sign and 24 bits and the exponent value, **n**, has 8 bits.

### **\$5,637,144,576\_{10}\$ in IEEE 754**

$5,637,144,576_{10} = 1\ 0101\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2$

Normalization:

$= 1.0101\ 0000\ 0000\ 0000\ 0000\ 0000_2 \times 2^{32}$

1. The exponent is  $32_{10} = 100000_2$  but in the IEEE 754 format we must add  $127_{10} = 1111111_2$  to this to give  $10011111_2 (=159_{10})$ .

2. In 32 bits this becomes:

**0100 1111 1010 1000 0000 0000 0000 0000**

Note that the normalization means that the m.s.b. of the A value is always 1 so this can be omitted.

The **m.s.b.** of the 32 bit word is a **sign bit**.





# Lecture 4

## Subroutines

So instead of  
this:

Instruction 1  
Instruction A  
Instruction B  
Instruction C  
Instruction 2  
Instruction 3  
Instruction A  
Instruction B  
Instruction C  
Instruction 4

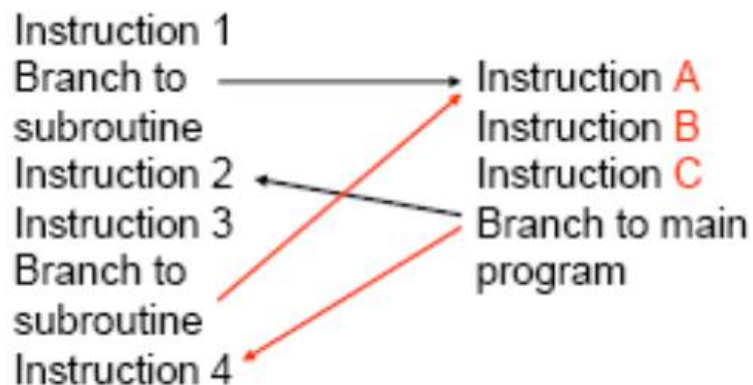
We have this:

Main program

Instruction 1  
Branch to  
subroutine  
Instruction 2  
Instruction 3  
Branch to  
subroutine  
Instruction 4

Subroutine

Instruction A  
Instruction B  
Instruction C  
Branch to main  
program



Many microprocessor programs include groups of instructions which are repeated many times. They can be included once in a special structure known as a subroutine.

The main program branches to the start of the subroutine when it requires those particular instructions.

At the end of the subroutine there is another branch back to the main program.

## Link Register

holds **the memory address** of the instruction in the main program **to which the subroutine returns to**.

Register `r14` in the ARM microprocessor is designated as the link register and `r14` can be replaced by `lr` in mnemonics.

## Branch and Link `BL`

To return from the subroutine the value in the link register is moved into the program counter

- `MOV pc, lr.`

What happens if a branch and link occurs in a subroutine?

- The value held in the link register will be overwritten by a new return address so before one subroutine calls another the link register value must be stored elsewhere.

## Nested subroutines

In order to preserve the return address of all subroutines an area of computer memory called the **stack** is used.

## Stack

**last in first out** queue.

### Stack Pointer

holds an address in memory that identifies the top of the stack.

The address is

- **full stack**: either the location of the last data to be pushed onto the stack
- **empty stack**: or alternatively the location of the next empty slot where the next data can be placed.

ARM microprocessor allows both kinds.

### Ascending and descending stacks

For an ascending stack, the memory address of the top of the stack is **greater** than the memory address for the bottom of the stack.

Register `r13` is used as the stack pointer and it can be replaced in mnemonics with `sp`.

```
STMFd sp!, {lr}
```

To push the link register value onto a full descending stack

```
LDMFD sp!, {lr}
```

To pop a value from a full descending stack back into the link register

## Stacks and Subroutines

1. The first instruction in a subroutine is

```
STMFd sp!, {lr}
```

2. Then any branch and link in that subroutine can overwrite the link register.
3. The subroutine would end by popping the return address from the stack into the link register.

```
LDMFD sp!, {lr}
```

4. followed by moving value in the link register into the program counter

```
MOV pc, lr
```

- replace these two instructions with one

```
LDMFD sp!, {pc}
```

## Stacking other registers

Pushing and popping the stack can be achieved with several registers at the same time

- the mnemonics `LDM` and `STM` stand for load multiple and store multiple.

```
STMFD sp!, {r6-r9, lr}
```

push registers r6, r7, r8 and r9 with the link register

```
LDMFD sp!, {r6-r9, pc}
```

pop the same registers

! Great care must be taken when using push and pop to **ensure that the number of pops is the same as the number of pushes.**

Order for Store: `r15, r14, ..., r0`

Order for Load: `r0, r1, ..., r15`

## Barrel shifter

a very useful feature of the ARM7 microprocessor which allows bit patterns to be rotated.

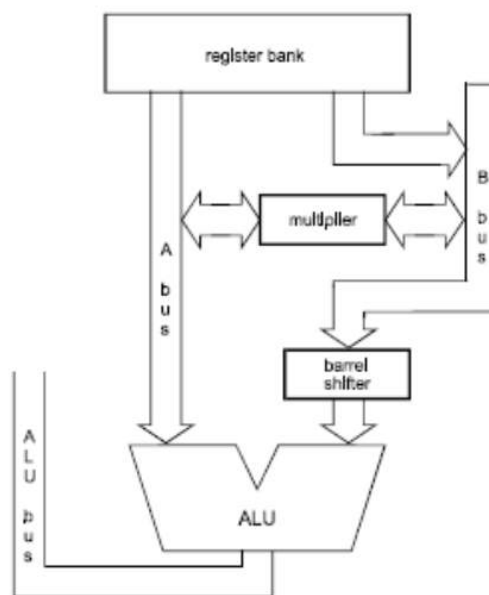
```
MOV r1, r2, LSL #5
```

take the bit pattern in register `r2` and shift it 5 places to the left before placing it in register `r1`.

Arithmetic Logic Unit: ALU

The ARM microprocessor divides the ALU functions into three blocks;

- a multiplier (that uses Booth's algorithm),
- the 'barrel shifter'
- and the rest of the ALU including the adder and logic functions. (This third block is generally referred to as the ALU.)



Different Shifts

- Logical shift left ( `LSL` )
- Logical shift right ( `LSR` )
- Arithmetic shift right ( `ASR` ).
  - bits are shifted rightwards and the new bits added in are the same as the 'old' sign bit; that is the msb of the input.
- Rotate right ( `ROR` )
- Rotate extended ( `RRX` )
  - bits are shifted right one place only and the carry flag is shifted into the new most significant bit. The least significant bit is shifted into the carry flag only if the mnemonic specifies an S.

## Using shift and add to multiply

`MOV rx, ry, LSL #n ;` multiplies value in `ry` by  $2^n$

`ADD rx, ry, ry, LSL #n ;` multiplies value in `ry` by  $2^{n+1}$

## Using shift to do integer division

`MOV rx, ry, ASR #n ;` divides value in `ry` by  $2^n$

...

# Lecture 5

## Interrupts

provide a very convenient method for dealing with events and this method is often used for events that are not unexpected

An interrupt is triggered when the microprocessor receives a (voltage) signal on a special connection within the control bus.

Two types of interrupts

- a normal interrupt (IRQ)
- a fast interrupt (FIQ)

## Interrupt Handling

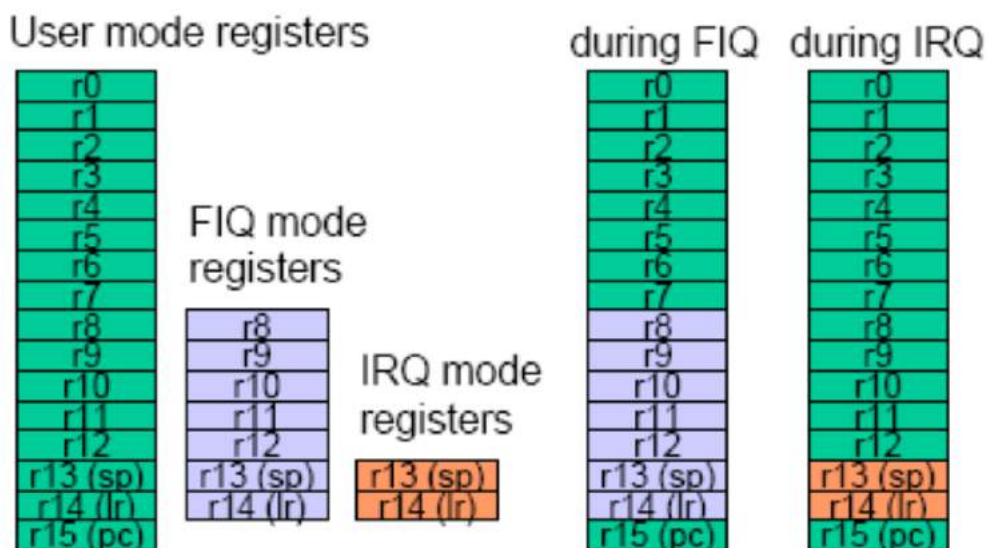
On receiving an interrupt signal, the microprocessor switches **mode**.

user mode, IRQ mode, FIQ mode

Each mode has its own link register and stack pointer.

In addition, FIQ mode has its own registers `r8` to `r12`.

## Registers in IRQ and FIQ modes



## Current Program Status Register

used in user-level programs to store the condition code bits

31	28							7	6	5	4	3	2	1	0	
N	Z	C	V	unused				I	F	T	mode					
											User:	10000				
											FIQ:	10001				
											IRQ:	10010				
											SVC:	10011				
											Abort:	10111				
											Undef:	11011				
											System:	11111				

### When an interrupt occurs the following happens:

1. The registers for IRQ mode or FIQ mode are activated.
2. The current program status register, **CPSR**, (this contains the flags and other information) is saved into a saved program status register, **SPSR**.

(There are two SPSR registers, one for each mode.)

3. The return address of the next instruction to be executed in the main program is stored in the link register for the appropriate mode.
4. The program counter is set to either **0x00000018** for an IRQ or **0x0000001c** for a FIQ. (these addresses are known as **vectors**)

For an IRQ the instruction at address **0x00000018** must be a branch to another part of memory because the following memory location, **0x0000001c** contains the first instruction of the FIQ handler.

### Returning from Interrupts

- The saved program status register is copied back into the current program status register.
- The link register in the IRQ mode or FIQ mode is copied into program counter.
- The microprocessor returns to the mode it was in before the interrupt occurred - normally user mode.

### FIQ or IRQ ?

Generally the most important interrupt is assigned to the FIQ and all other interrupts are assigned to IRQ.

Reasons:

1. IRQ is disabled by an FIQ and if a FIQ and an IRQ occur simultaneously the FIQ is serviced first.
2. FIQ can be serviced as quickly as possible because
  1. there is no need to branch as for an IRQ and
  2. normally user mode registers are pushed onto the stack when an interrupt occurs so that they are not corrupted but for an FIQ there is no need to stack registers r8 to r12.

## Instruction Pipelines

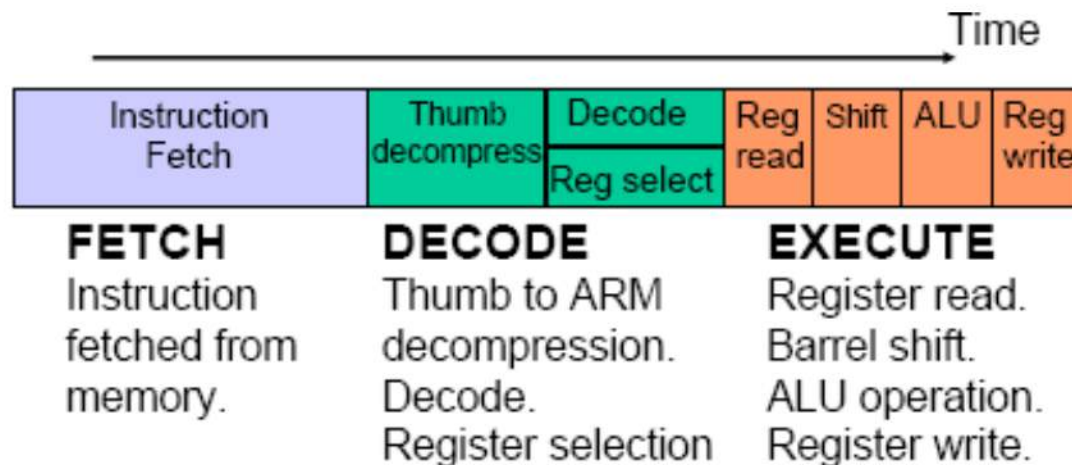
an important feature of all modern microprocessors.

an n stage instruction pipeline allows the microprocessor to execute up to n times as many instructions in a given time.

The ARM7 microprocessor has a 3 stage pipeline, one stage for each of the CPU cycles.

**fetch, decode and execute.**

In a three stage pipeline, the CPU can simultaneously execute an instruction, decode the next instruction and fetch the next instruction.



## Optimum Operation

A pipeline operates optimally if the instructions to be executed are in consecutive memory locations and no conflict occurs on the data bus.

When this is the case, the microprocessors operates at one clock cycle per instruction (1 CPI).

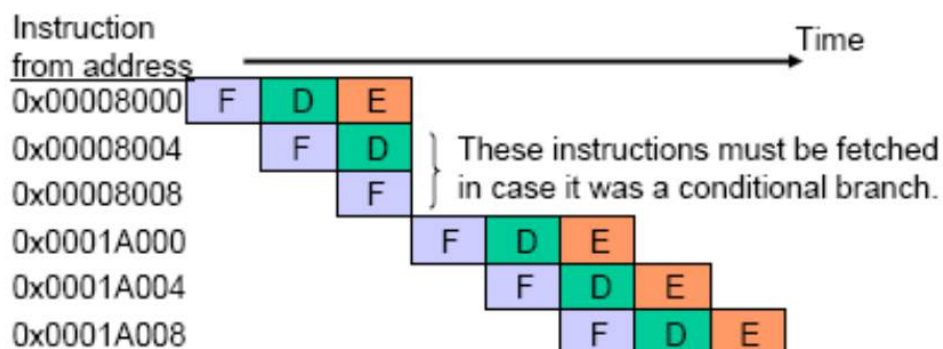
The best performance cannot be achieved if a branch or load instruction is executed or if an interrupt is serviced.

## Branch Instruction

A branch instruction will reload the program counter so that two cycles are lost.

If not executed then no clock cycles are lost.

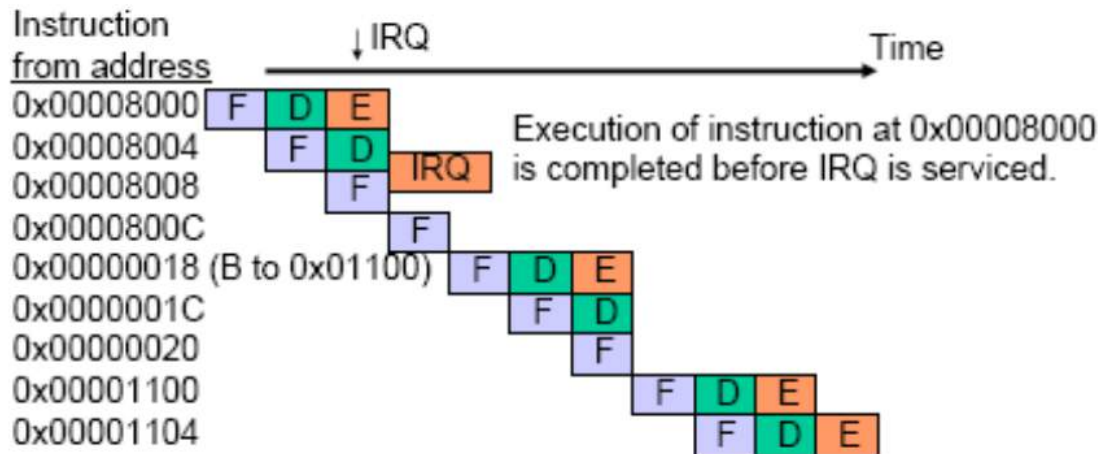
- e.g. assume that the instruction at address 0x00008000 is Branch to 0x0001A000



## Interrupts



An IRQ reloads the program counter with `0x00000018` and then branches so that the pipeline is broken twice.



## Interrupt latency

the time between the microprocessor receiving an interrupt signal and the first instruction being executed.

The minimum latency for an IRQ is 7 clock cycles.

minimum because an IRQ could be interrupted by an FIQ.

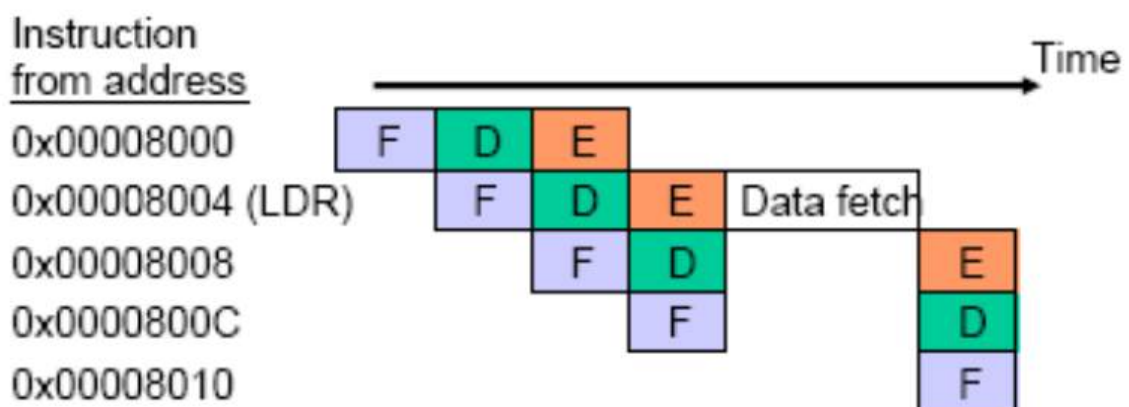
The minimum latency for an FIQ is 4 clock cycles.

no need to branch

FIQ can be interrupted by a system reset.

## Load and store instructions

use the data bus to pass data to or from memory so that the data bus cannot be used to fetch an instruction at the same time.



## Eliminating bus conflicts

- Harvard architecture

uses two separate data buses; one for instructions and one for load and store data.

- von Neumann architecture

(for a microprocessor, such as the ARM7)

uses one data bus for both instructions and data.

### **Von Neumann or Harvard?**

- advantage: fewer lost clock cycles due to bus conflicts
- drawback: greater complexity

The ARM9 (next generation after ARM7) has a Harvard architecture and a five stage instruction pipeline.

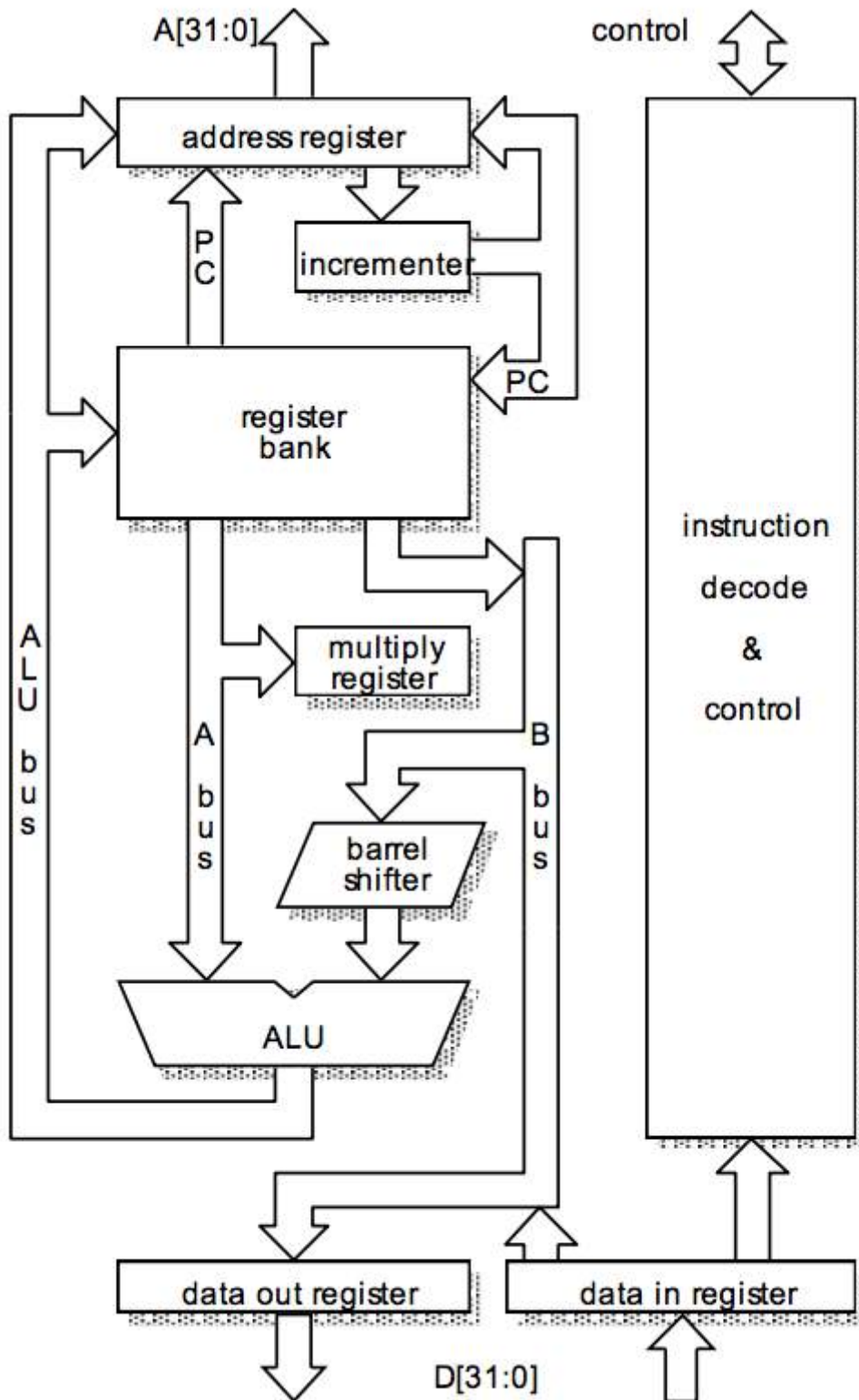
### **3 stage or 5 stage pipeline?**

The maximum clock frequency of the ARM7 is limited by a bottleneck during the execute stage of the 3 stage pipeline.

The ARM9 has a 5 stage pipeline so that there is less work in each stage of the pipeline and therefore a higher maximum clock frequency can be achieved for the same technology.

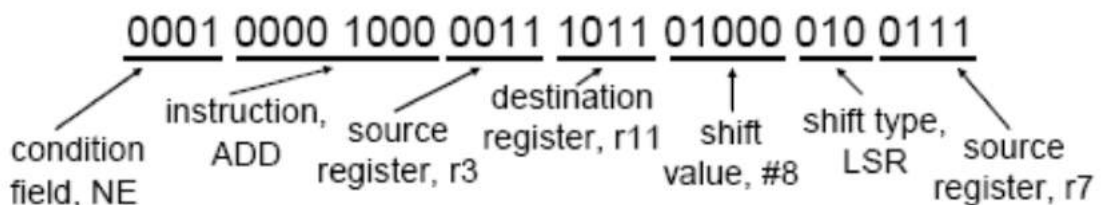
However more stages require more power because more circuits are functioning simultaneously.

### **ARM organization**



## ARM7 pipeline example

ADDNE r11, r3, r7 LSR #8



## FETCH

The instruction is fetched from a memory location with address given by the value in the address register.

## DECODE

The first action of the decode cycle only applies to 16 bit Thumb instructions. Thumb code is converted (decompressed) into 32 bit ARM code if the processor is in 'Thumb mode'.

In 2nd part of the decode cycle, the instruction, is decoded.

The condition field is compared with flags to see if the instruction is to be executed.

## EXECUTE

- Register read.

The values held in the two source registers are read onto the internal buses

- Barrel shift.

Output of the barrel shifter is connected to the second input of the ALU.

- ALU

The output of the ALU is connected to the ALU bus.

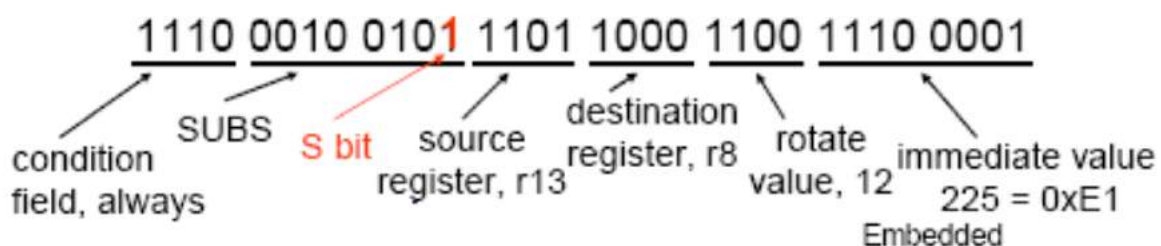
- Register write.

The value on the ALU bus is written into the destination register.

ALU is not used to increment the program counter and address register so that it is free to execute instructions.

## Another example: immediate

```
SUBS r8, r13, #57600
```



The ARM7 barrel shifter only supports 'rotate right', not 'rotate left'. This is not a problem because a rotate left by x bits is equivalent to a rotate right by (32 - x) bits.

## ARM core: not pure RISC

ARM cores do not implement all of the features proposed by Patterson and Ditzel. The features included are:

1. Load-store architecture;
2. Fixed length 32-bit instructions with few formats;
3. Hardwired combinational decode logic;
4. Pipelined execution.

The rejected features are:

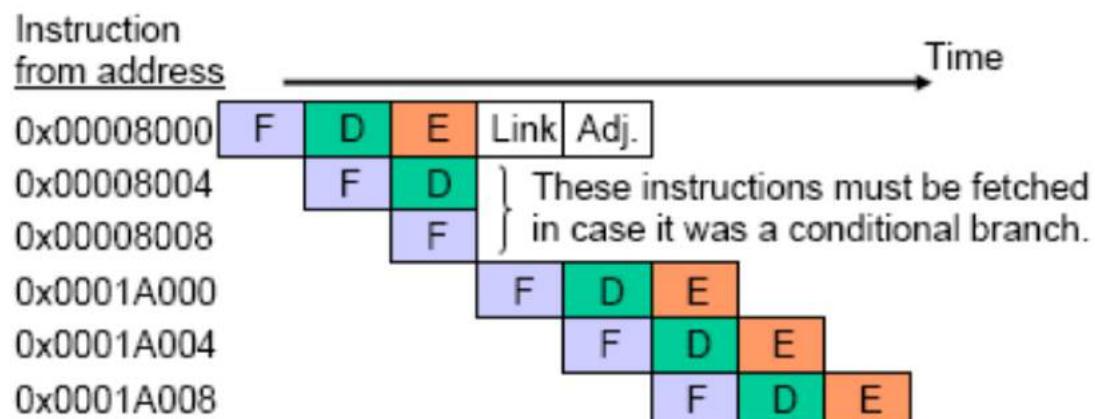
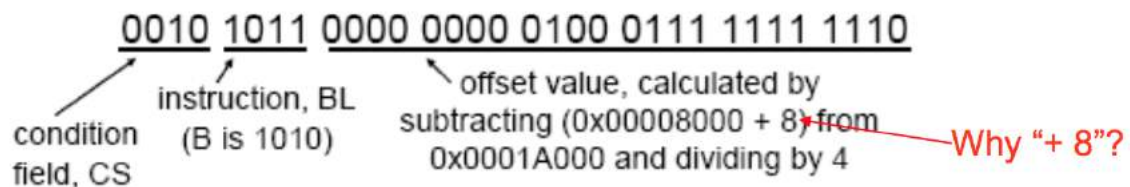
1. The large register bank, the ARM only has 16 registers;
2. Delayed branches;
3. Single-cycle execution of all instructions.

## ARM: multiple cycle instructions

1. Multiplication – could be executed in a single cycle but would need a disproportionately large no. of gates;
2. Memory data access – there needs to be two memory accesses, one for the instruction & one for the data, so the ARM7 uses 2 cycles. Other processors use 2 buses, 2 caches or 2 memories i.e. Harvard architecture. ARM uses extra cycle for features e.g. auto-indexing.
3. Branches – result in pipeline flush. Wasted cycles used to update 'link register'.

## ARM7 branch example

BLCS Table



3 cycles of EXECUTE

1. ALU adds two values, ( 0x00008008 & 0x00011FF8 ) to find a new memory address ( 0x0001A000 ) to be loaded into the address register.
2. Value ( 0x00008008 ) in r15 (PC) is loaded onto the A bus and passed through the ALU (no action) to r14

3. Value in `r14` (LR) is adjusted by subtracting 4 in the ALU & the new value `0x0008004` is stored back in `r14`. `r15`, the PC, is brought up-to-date.

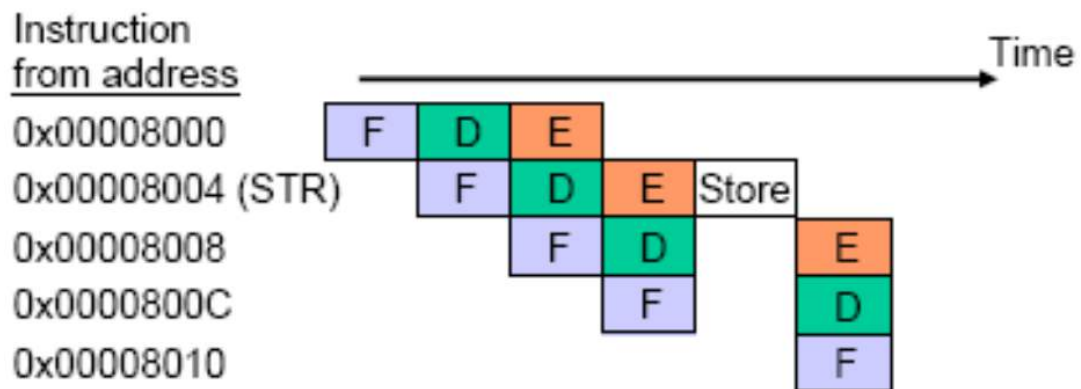
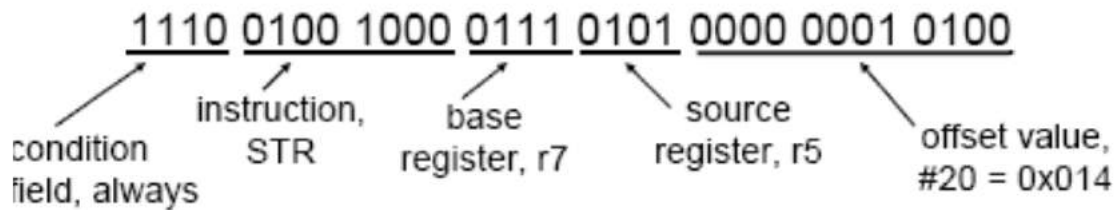
???

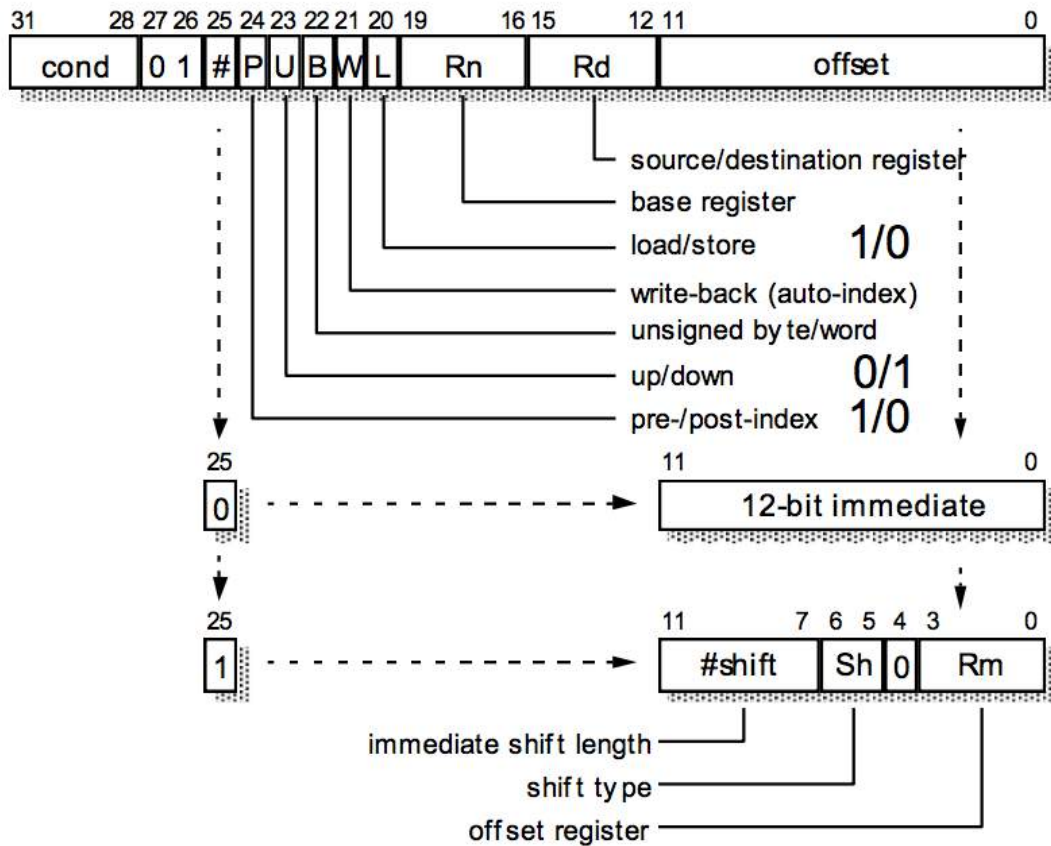
The link register must be adjusted because the program counter is 8 ahead of the instruction being executed, whereas the return address is only 4 ahead.

## ARM7 pipeline and Store

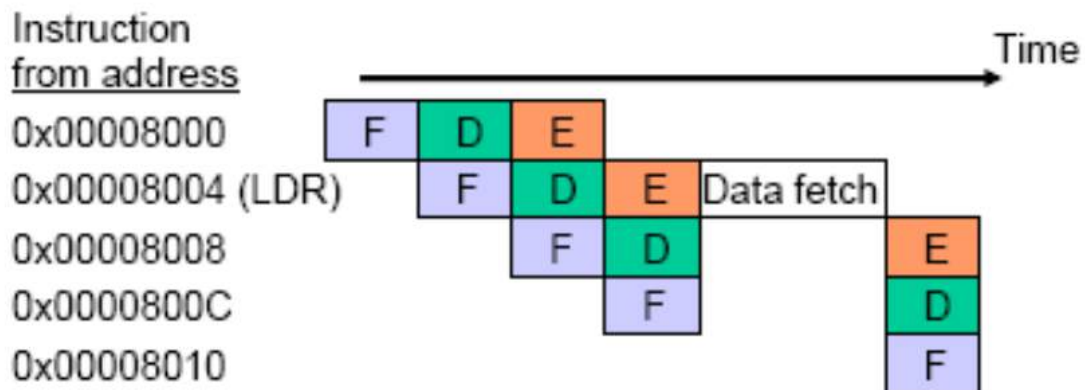
`STR r5, [r7], #20`

post-indexing

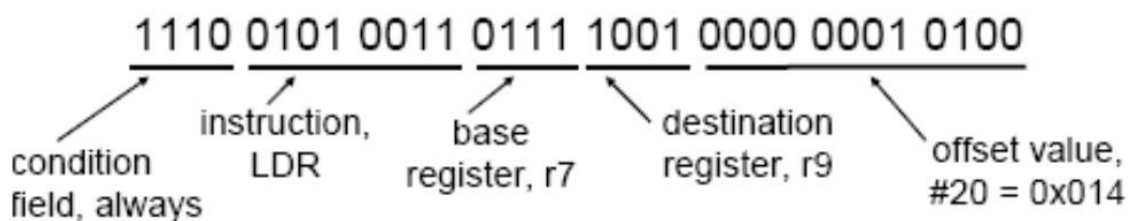




## ARM7 pipeline and Load



LDR r9, [r7, #-20]!



Load and store have many options e.g.

1. offset by either immediate or register,
2. positive or negative offset (or zero),



3. pre-indexing or post-indexing of base register,
4. automatic updating, 'pling', of base register (pre-indexing only),
5. byte or word.

## Why provide auto indexing?

Auto-indexing saves one clock cycle (and power dissipated in memory) by using spare resource during the load/store.

## Further cycle saving

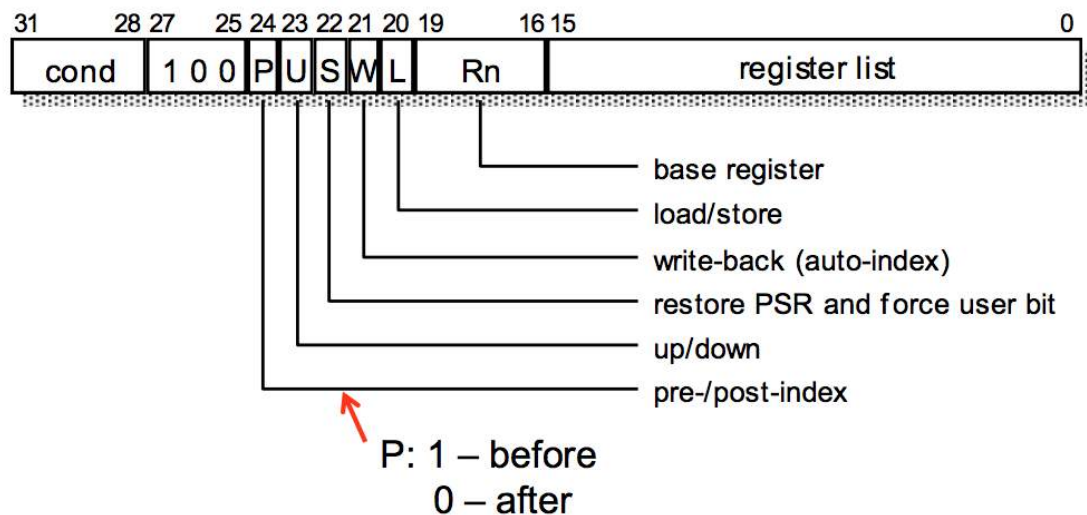
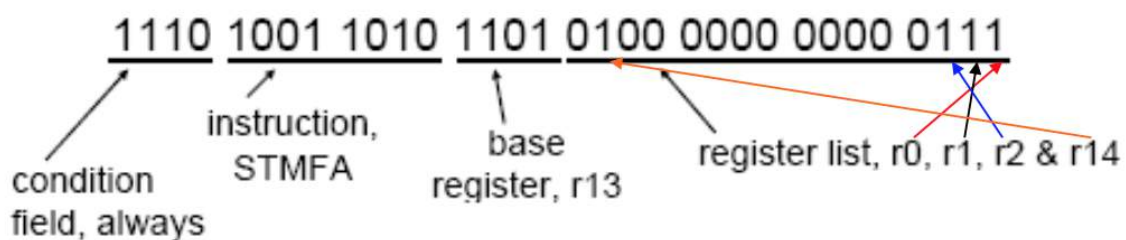
More clock cycles can be saved if more than one register is loaded or stored by a single instruction.

LDM takes  $(2 + n)$  clock cycles to execute where  $n$  is the number of registers loaded.

Likewise STM takes  $(1 + n)$  clock cycles.

## ARM7 Store Multiple example

`STMFA r13!, {r0-r2, r14}`



## ARM7 performance

ARM7 does make efficient use of the data bus with nearly every clock cycle used for either an instruction pre-fetch or a data load/store.

The data bus is only idle during

- the last cycle of load (LDR/LDM)
- all but the first cycle of a multiply (MUL/MLA) instruction.



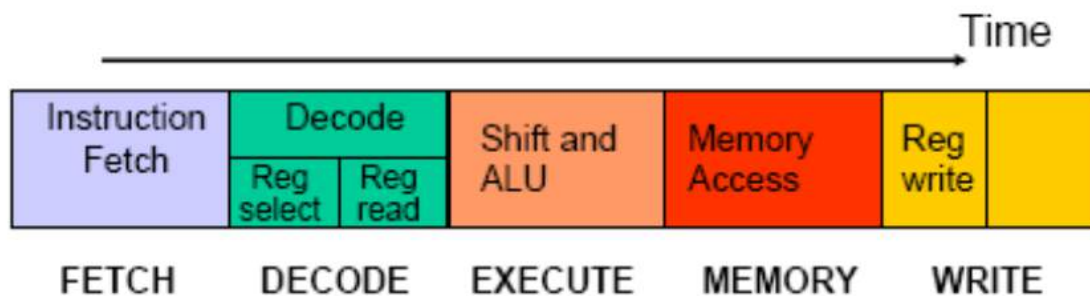
## Performance improvements

Performance improvements can be gained by including:

- More pipelining i.e. longer pipeline
- Harvard architecture
- Delayed branches
- Additional specialized instructions

### Longer pipelines

The ARM9 5 stage pipeline is more balanced because the 'execute' functions are spread over two and a half pipeline stages called `execute`, `memory` and `write`.



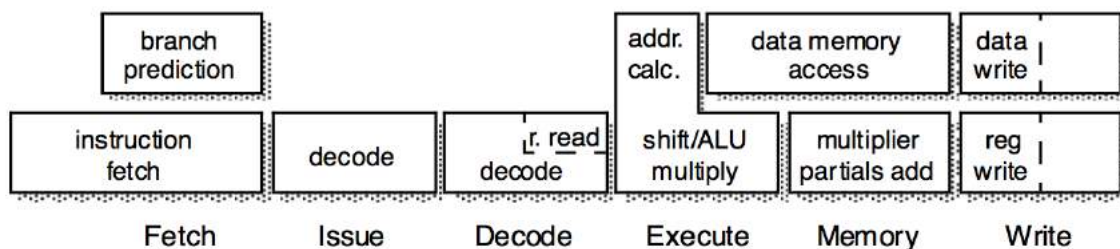
Register read is moved into the decode stage and register write is delayed until after a separate stage for any memory data access (load/store).

More stages = less activity  
= higher clock frequency  
= higher performance

### ARM9 optimal pipelining

(See slides)

### ARM10: 6 stage pipeline





# Lecture 6

---

## Memory

---

a device that can store information in a binary format.

### Read-only memory (ROM)

used for the operating system on a desktop computer or an application program in an embedded system such as a mobile phone.

### Read-write memory (RAM)

used for an application program on a desktop computer and for temporary data.

### Types of ROM

- Programmable ROM (PROM) could be electrically programmed once only and then the contents were fixed.
- Erasable PROM (EPROM) could also have its contents erased by exposure to ultraviolet (UV) light and could then be reprogrammed.
- Electrically erasable PROM (EEPROM) could have its contents erased by an electrical signal.
- Flash Erasable PROM (FEPRM or flash memory) is similar to EEPROM but erasing could only be done over a significant part of (maybe all) the integrated circuit.

**All ROM is non-volatile** - meaning the information stored in ROM memory is not lost when it is disconnected from a power source.

In contrast the contents of RAM memory will be lost almost as soon as power is switched off - **RAM is volatile**.

## RAM

Random Access Memory

the time taken to read or store data from or to the memory (the 'access time') does not depend upon the order in which the data is accessed.

The memory locations can be accessed randomly or sequentially in the same time.

This is not true for hard disks for example.

There are two types of RAM

- static
- dynamic

## Dynamic RAM

---

The contents of dynamic RAM (DRAM) **must be refreshed every few milliseconds**.

The binary data is held as an electrical charge on a tiny capacitor

- a charged capacitor represents a binary 1
- whereas an discharged capacitor represents a binary 0.

However the charge on the capacitor can leak away so it must be topped up or refreshed every few milliseconds.

## Static RAM

---

Static RAM (SRAM) uses D type latches to store data and **it does not need to be refreshed**.

The data stored in a SRAM memory will be retained as long as it is connected to a power supply.

SRAM is **generally faster than DRAM** but needs a greater surface area of silicon (means **more expensive**) to store the same amount of data.

## Memory: fast or big?

---

In general a bigger memory is a slower memory.

The main memory of a microprocessor is typically DRAM.

## Memory Cache

---

a small fast on chip memory that holds the most recently accessed data from the main memory.

cache hit / miss

- **Temporal locality** occurs because data accessed once is likely to be accessed again soon
- **Spatial locality** occurs because data from one memory location is likely to be accessed if data in an adjacent memory location has recently been accessed.

## hit rate

should be over 90% to achieve the best results with modern microprocessors.

Sometimes there may be two levels of cache

- 'primary cache' that is on chip
- off chip 'secondary cache'.

Also there may be separate caches for instructions and data, like the 'modified Harvard' architecture.

HIT ratios over 0.95 (95% hits) can be obtained by good cache design.

$H$  will depend upon the program running.

Miss ratio,  $M$

## Mean access time

---

$$t_{ave} = h \times t_c + m \times t_m = h \times t_c + (1-h) \times t_m = (1-m) \times t_c + m \times t_m$$

- In practice the cache control and routing circuits will add an extra time delay of ***a***.

$$\begin{aligned} t_{ave} &= h \times t_c + m \times t_m + a = h \times t_c + (1-h) \times t_m + a \\ &= (1-m) \times t_c + m \times t_m + a \end{aligned}$$

## Multiple levels of cache

Very complex multiple cache systems are used with high performance CPUs, these have multiple caches operating at different levels.

## Cache control systems

Two main designs:

- associative cache
- direct mapped cache.

block = several bytes of main memory copied to one cache line.

### Associative cache

## Associative cache organization

Cache line number ↓	Valid bit	Block number	Block contents
0			
1			
2			
3	1	10110010	0100101001001000111101....
last			
No. of bits →	1	$Y - a$	$8 \times (2^a)$

Cache line length is =  $(1 + Y - a + 8 \times 2^a)$  bits

8 bits per byte

number of bytes

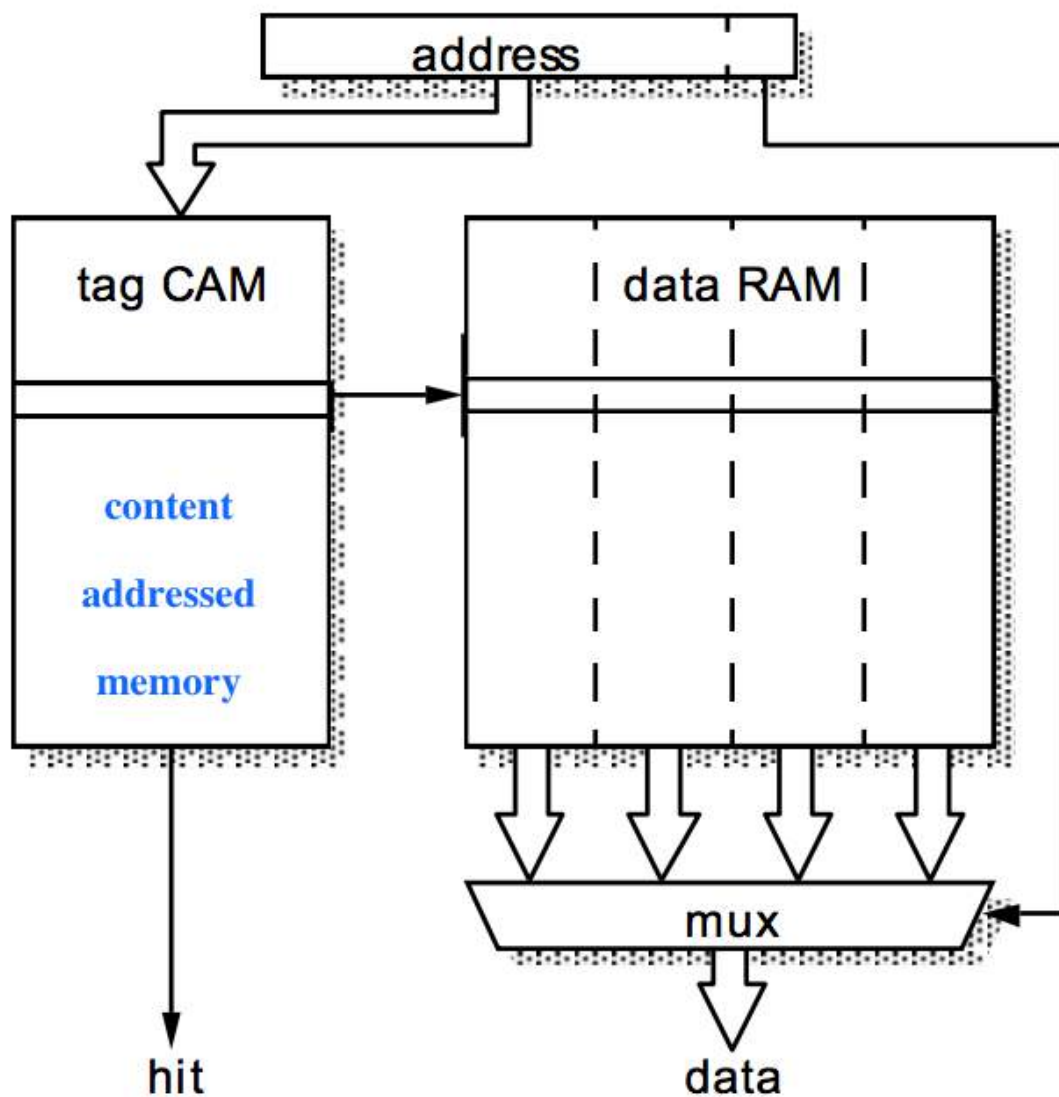
- Number of bytes in a block is  $X$  with  $X=2^a$  where  $a$  is a fixed integer value, typically 4.
- The block number is the memory address right shifted  $a$  places so that if the main memory address has  $Y$ -bits, block number has  $Y - a$  bits.
- The cache memory has a very large word length - line length - longer than memory word length.
- Each cache line has a valid bit (tag) to show if the line holds a valid copy of main memory words (all set invalid at power up).

## Fully associative cache

In this example the number of bytes stored in a cache line is 16. The data byte at main memory address 0x37B6A038 is 0xD4 – highlighted.

Tag field	Block contents
0x9AB6301	0xC2A1096C 6FE3D674 A956410B D4FE3D67
0x54B09FF	0xD4FE3D67 A956410B 6FE3D674 C2A1096C
0x37B6A03	0x6FE3D674 C2A1096C D4FE3D67 A956410B
0xF31A004	0xC2A1096C D4FE3D67 6FE3D674 A956410B
0xFF4E502	0x6FE3D674 C2A1096C A956410B D4FE3D67
0x07D3A00	0xA956410B C2A1096C 6FE3D674 D4FE3D67
...	... ..
0x96301AB	0x6FE3D674 C2A1096C D4FE3D67 A956410B

0                      8                      15

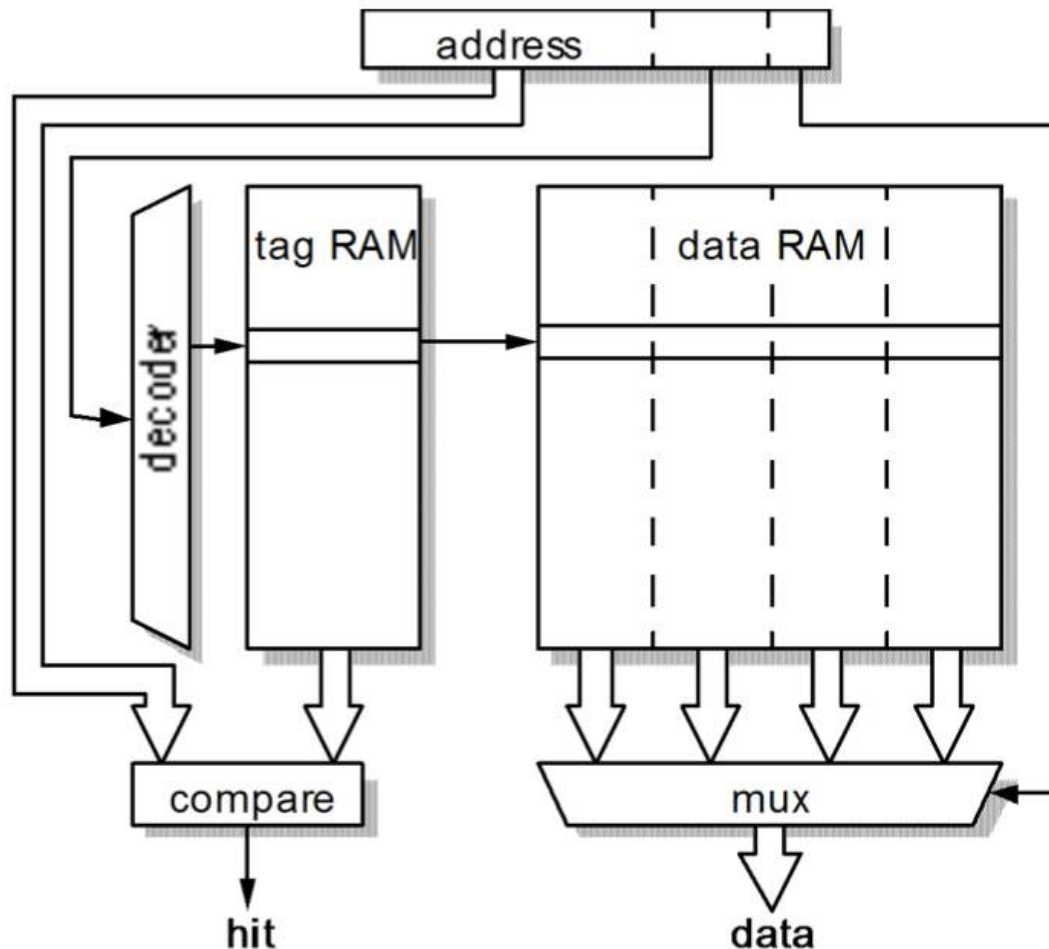


### Direct-mapped cache

Cache line number ↓	Valid bit	Tag field	Block contents
0			
1			
2			
3	1	01011010	<u>01001010</u> <u>001001000</u> <u>1111010</u> ...
$2^M-1$			
No. of bits →	1	$Y - M - a$	$8 \times 2^a$

So if  $M = 4$  and  $a = 4$  the contents of main memory address 01011010 0011 0000 and the following  $2^4 - 1$  addresses is 01001010 01001000 1111010.....

- Avoids cache search by relating cache line number to the main memory address.
- There are  $2^M$  lines in the cache with  $M$  bits used to identify each cache line.
- Main memory address is divided into three parts, the least significant  $a$  bits are discarded as each cache line contains 2 bytes.
- The next  $M$  bits are used as the cache line number - so several different main memory addresses correspond to a single line in the cache.
- The most significant  $(Y - M - a)$  bits are used to determine which address is in cache and these are stored in the cache line as a tag field.



## Write access problem

Two basic solutions –

- write through
  - every cache write, simultaneously writes to main memory.
- copy back.
  - cache line has an extra bit that is cleared on loading, if a write is performed, it is set.

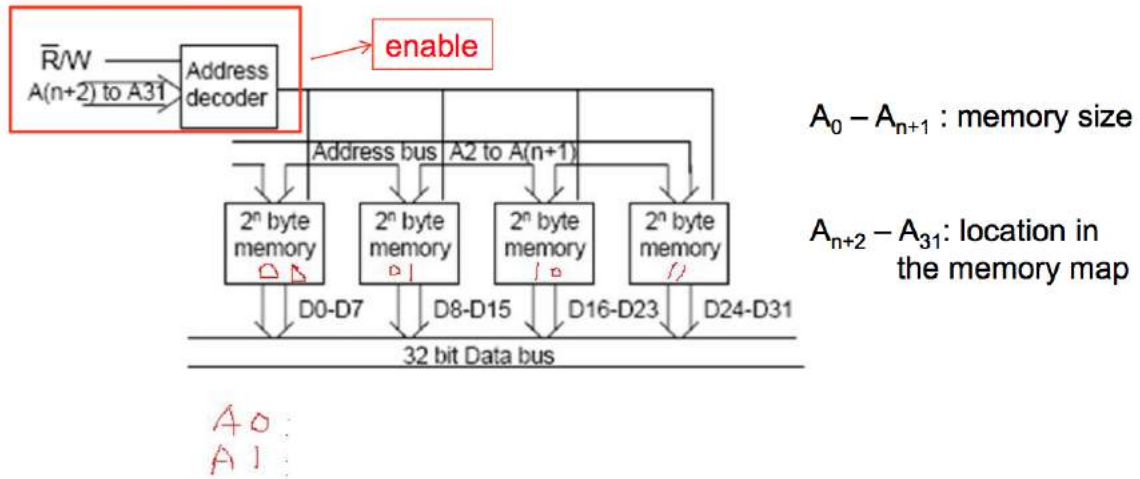
A minor additional decision is the action on a **write miss**, simple system is to put into cache, an alternative is to only alter main memory.

Simple 4 byte ROM addressing

Connecting to a 32 bit data bus



- Memory with 8 data bits output can be connected to a 32 bit data bus by using 4 separate memory chips; one for each byte



## Input and output

Input to and output from a microprocessor can be arranged in two ways:

– either as an additional subsystem with dedicated hardware – or as part of the memory system.

Many microprocessors, include the ARM, use ‘memory mapped’ input and output.

Not all of this ‘addressable space’ will be filled with actual memory so that there are many ‘empty’ memory locations.

Memory mapped input ports and output ports are assigned a memory address and

- A register load from that address is equivalent to an input.
- A register store to that address is equivalent to an output.

## Peripheral Communication

There are two basic communication techniques:

- Polling
  - programming the processor to repeatedly check to see if the communication task has been completed or not
- Interrupts
  - either the processor issues commands to the peripheral to start communication task, then wait for an interrupt to signal completion of the task, or the processor waits for an interrupt from the peripheral to start communication task.

## The Device Driver Philosophy

The benefits of good device driver design are threefold:

- First, because of the modularization, the structure of the overall software is easier to understand;
- Second, because there is only one module that ever interacts directly with the peripheral’s registers, the state of the hardware can be more accurately tracked
- Last but not least, software changes that result from hardware changes are localized to the device driver.

### Device Driver Implementation

1. A data structure that overlays the memory-mapped control and status registers of the device. To make the bits within the control register easier to read and write individually, we might also define bitmasks such as:

```
#define TIMER_ENABLE    0xC000 // Enable the timer
#define TIMER_DISABLE   0x4000 // Disable the timer
#define TIMER_INTERRUPT 0x2000 // Enable timer interrupt
```

2. A set of variables to track the current state of the hardware and device driver;

3. A routine to initialize the hardware to a known state;

4. A set of routine that, taken together, provide an API for users of the device drive, such as sending and receiving messages, checking whether it is running correctly (health check), etc.;

5. One or more interrupt service routines.

...

# Lecture 7

---

## How do you make a computer?

---

- Input and output devices
- Memory (for both data and instructions)
- A CPU that can be further broken down into
  - 1) control circuits
  - 2) instruction decoder
  - 3) arithmetic and logic circuits (ALU).

Furber differentiates between computer architecture and computer organization

Computer architecture describes the user's view of the computer. The instruction set, visible registers, memory management table structures and exception handling model are all part of the architecture.

Computer organization describes the user- invisible implementation of the architecture. The pipeline structure, transparent cache, table-walking hardware and translation look-aside buffer are all aspects of the organization.

The core component is the CPU which contains

- control circuits
- registers
- instruction decoding circuits
- an arithmetic and logic unit (ALU)
- an oscillator or clock to drive the system

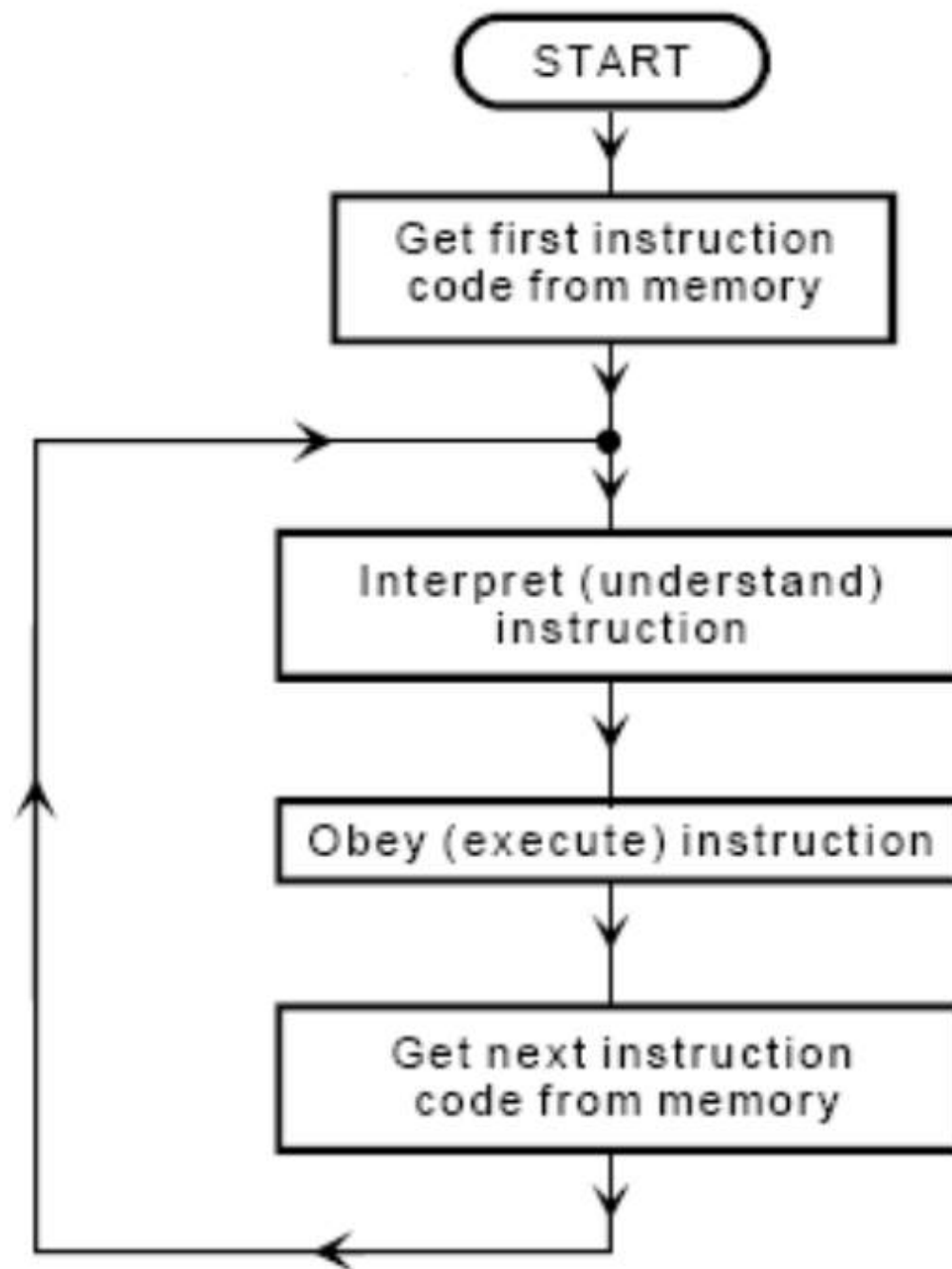
## Von Neumann systems

---

The basic von Neumann architecture/structure has

- Memory for storing both data and instructions
- Control and decode
- Arithmetic unit (now arithmetic and logic unit, ALU)
- Input and output, IO, mechanisms

## The von Neumann cycle

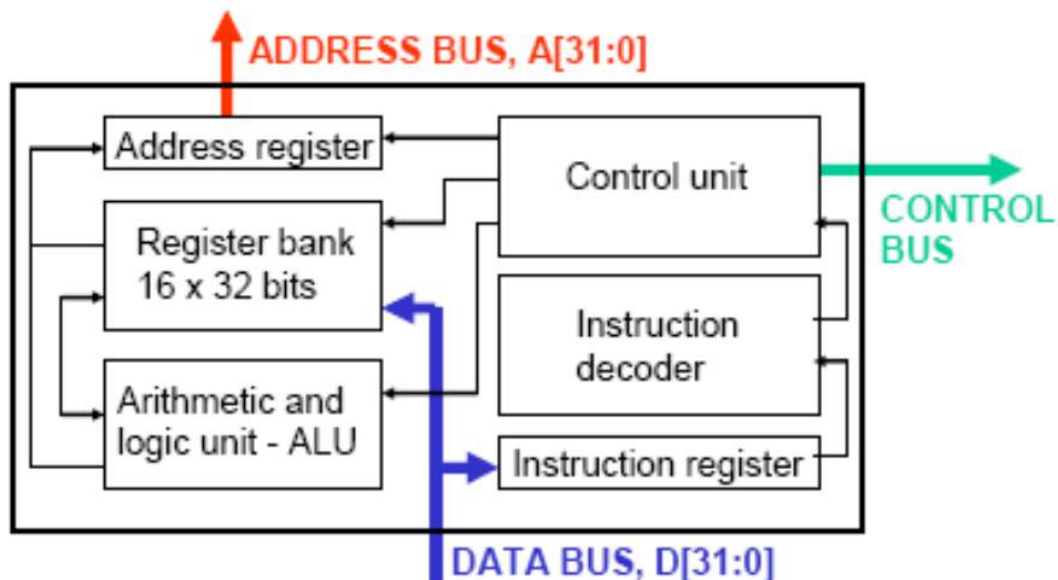


## MU0 – A simple microprocessor

(See slides)

## ARM7 CPU architecture

---



## ALU circuits: design problem

### Combinational logic circuit design

Design of large combinational (e.g. arithmetic) circuits  
 – some approaches

- modular methods (e.g. cell technique)
- replace by connected more simple modules used in sequence – convert to a sequential circuit.

### Modular - the cell technique

- break the circuit into separate modules – cells
- the chosen cells are small enough to be designed and built
- if possible only a small number of different types of cell are used.
- the cells are interconnected to form the complete circuit.

### ALU adder design

A half-adder for the LSB and a full-adder for the other bits.

Connect the cells to give a ripple-carry adder.

Problem with ripple-carry adder

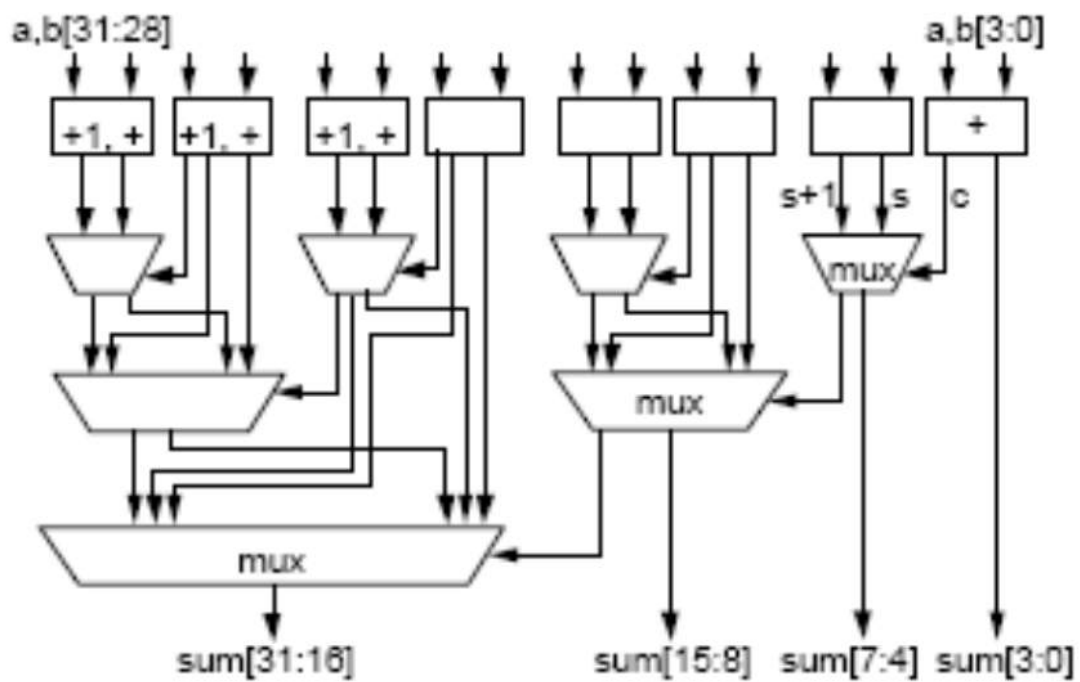
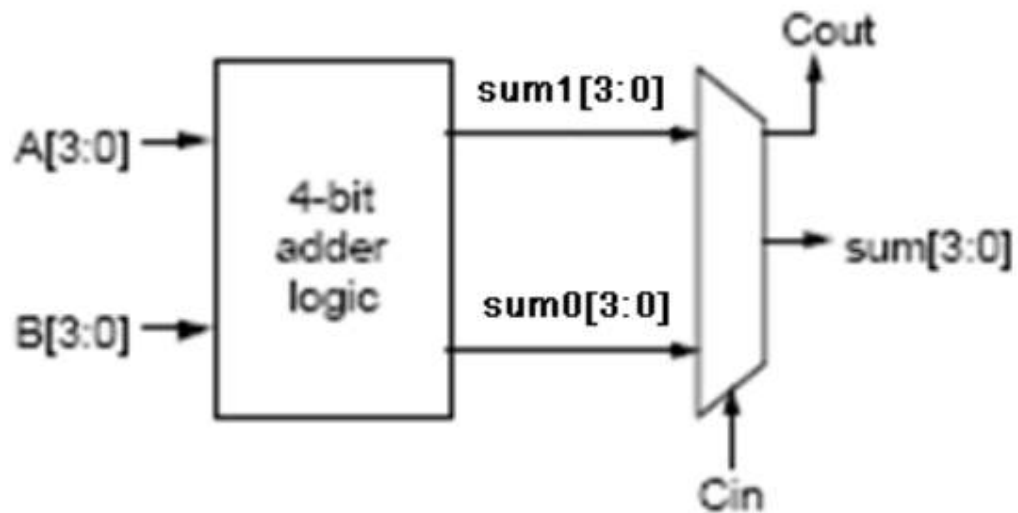
- Each cell causes a propagation delay
- For an adder with many bits, the delays become very long.

### ARM1 ripple-carry adder circuit

### ARM2 4-bit carry look-ahead

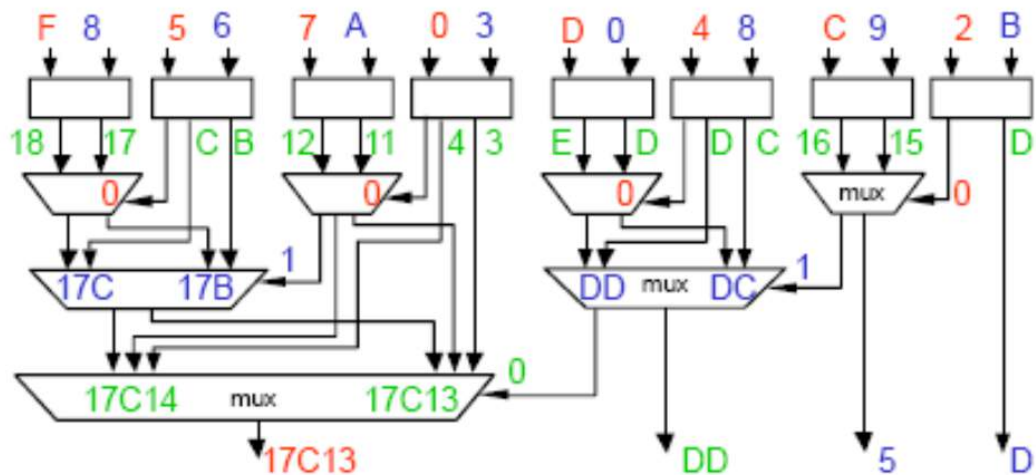
8 propagation delays for a 32 bit adder.

### ARM6 carry-select adder



For a 32 bit adder there are a **maximum of 3 propagation delays** in the carry path.

For example adding 0xF570D4C2 to 0x86A3089B:



The sum is 0x7C13DD5D with Cout = 1.

### Performance comparison

Size of adder	Ripple carry	Look ahead	Carry select
4 bits	4	1	1
8 bits	8	2	1
16 bits	16	4	2
32 bits	32	8	3
64 bits	64	16	4

The performance improvement is at the expense of more complicated circuits using more electrical power.

### Other arithmetic functions

- Subtraction
  - using twos complement, an adder will perform subtraction. Or use ones complement and set 'carry-in' to logic 1.
- Multiplication
  - complicated, consider  $0110 \times 1001$

### Multiplication circuits

A matrix multiplier for 32-bit numbers needs  $32 \times 32$  AND gates and 31 (32 bit) adders.

This would require **a large number of logic gates** and it would suffer from the problem of **multiple propagation delays on the carry path**.

Most multiplier circuits use sequential logic.

## Sequential approach

In simple form it requires double length registers and a shifter.

1. Set a total to zero.
2. Examine the LSB of the multiplier, if it is 1 add the multiplicand to the total otherwise do nothing.
3. Shift the multiplicand one place left moving in 0 as LSB.
4. Shift the multiplier one place right discarding the old LSB and moving in zero.
5. Repeat from 2 until all bits of multiplier tested (alternatively can end when multiplier is zero).

Problem: To perform a 32 bit multiplication would need 32 steps i.e. 32 clock cycles – too long.

## An improved rule (algorithm) - Booth's algorithm

This is another possible sequence using an existing adder and a shifter.

### Booth's multiplication algorithm

#### Initialization

1. Set a running total,  $T$ , to zero
2. Put the multiplier in a special register  $M$
3. Set a carry bit,  $C_{in}$ , to 0
4. Set a cycle counter,  $N$ , to 0

#### THIS is the LOOP ENTRY point

5. Take the two LSBs of  $M$ , call this value  $B$  and add value of  $C_{in}$  to  $B$
6. **Next action is one of**
  - i. if  $C_{in} + B = 000_2$ , do nothing and set  $C_{out} = 0$ .
  - ii. if  $C_{in} + B = 001_2$ , left shift the multiplicand  $2N$  places & add this to running total  $T$ , set  $C_{out} = 0$
  - iii. if  $C_{in} + B = 010_2$ , left shift the multiplicand  $2N+1$  places & add this to running total  $T$ , set  $C_{out} = 0$
  - iv. if  $C_{in} + B = 011_2$ , left shift the multiplicand  $2N$  places & **subtract** this from running total  $T$ , set  $C_{out} = 1$
  - v. if  $C_{in} + B = 100_2$ , do nothing and set  $C_{out} = 1$ .
7. Add 1 to  $N$  and copy  $C_{out}$  to  $C_{in}$ .
8. Right shift  $M$  two places (discarding bits shifted out)
9. If not dealt with all bits of multiplier go back to step 5.

The multiplication can end when only zeros are left in the multiplier.

Example:



$$100_{10} \times 743_{10} = \dots 0000\ 0110\ 0100_2 \times \dots 0010\ 1110\ 0111_2$$

N	C <sub>in</sub>	Multiplier	LSL	ALU	C <sub>out</sub>	Running total
0	0	$\times 11_2$	#0	A-B	1	1111 1111 1111 1111 1111 1111 1001 1100
1	1	$\times 01_2$	#3	A+B	0	0000 0000 0000 0000 0000 0010 1011 1100
2	0	$\times 10_2$	#5	A+B	0	0000 0000 0000 0000 0000 1111 0011 1100
3	0	$\times 11_2$	#6	A-B	1	1111 1111 1111 1111 1111 0110 0011 1100
4	1	$\times 10_2$	#8	A-B	1	1111 1111 1111 1111 1001 0010 0011 1100
5	1	$\times 00_2$	#10	A+B	0	0000 0000 0000 0001 0010 0010 0011 1100

### Booth's algorithm for negative numbers

Booth's multiplication algorithm also works with negative numbers in the two's complement format.

For instance multiplying a positive number,  $x$ , by  $-y$ . In 32 bit two's complement format  $-y$  is represented by  $(2^{32}-y)$  so that the product is  $(x.2^{32}-x.y)$ . Taking the lowest 32 bits only, the result is  $-x.y$ .

Likewise multiplying  $-x$  by  $-y$ , the product is  $(2^{32}-x).(2^{32}-y) = (2^{64}-(x+y).2^{32}+x.y)$  which is  $x.y$  when only the lowest 32 bits are considered.

The problem of 64 bit results will be considered later.

Example:

$$(-100) \times (-200) = (11\dots 11\ 1001\ 1100_2) \times (11\dots 11\ 0011\ 1000_2)$$

N	C <sub>in</sub>	Multiplier	LSL	ALU	C <sub>out</sub>	Running total
0	0	$\times 00_2$	-	A+0	0	0000 0000 0000 0000 0000 0000 0000 0000
1	0	$\times 10_2$	#3	A+B	0	1111 1111 1111 1111 1111 1100 1110 0000
2	0	$\times 11_2$	#4	A-B	1	0000 0000 0000 0000 0000 0011 0010 0000
3	1	$\times 00_2$	#6	A+B	0	1111 1111 1111 1111 1110 1010 0010 0000
4	0	$\times 11_2$	#8	A-B	1	0000 0000 0000 0000 0100 1110 0010 0000
5	1	$\times 11_2$	-	A+0	1	0000 0000 0000 0000 0100 1110 0010 0000
....	1	$\times 11_2$	-	A+0	1	0000 0000 0000 0000 0100 1110 0010 0000
15	1	$\times 11_2$	-	A+0	1	0000 0000 0000 0000 0100 1110 0010 0000

Using just the existing adder and barrel shifter a 32 bit multiplication can be completed in 16 cycles using Booth's algorithm.

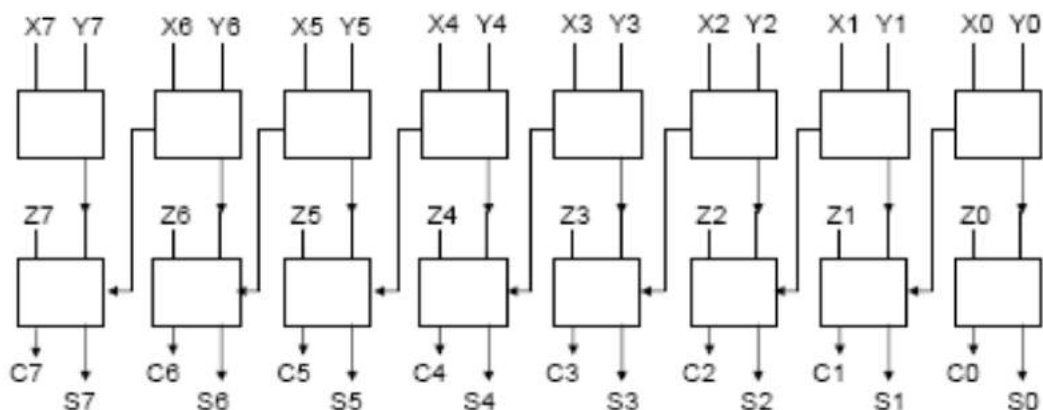
## Further performance improvements

The ARM uses a special adder circuit that adds five 32 bit numbers together in one combinational logic circuit.

Carry save adder

### $2 \times 8$ bit 'carry save' adder

The 'carry save' structure shown below. The sum  $(X+Y+Z)$  can be found by adding the partial sum  $S$  to the partial carry  $C$ . The total propagation delay is greatly reduced.



The ARM high performance multiplier uses a  $4 \times 32$  bit carry save adder to add six numbers, a running total (partial sum and partial carry) and four values produced using Booth's algorithm.

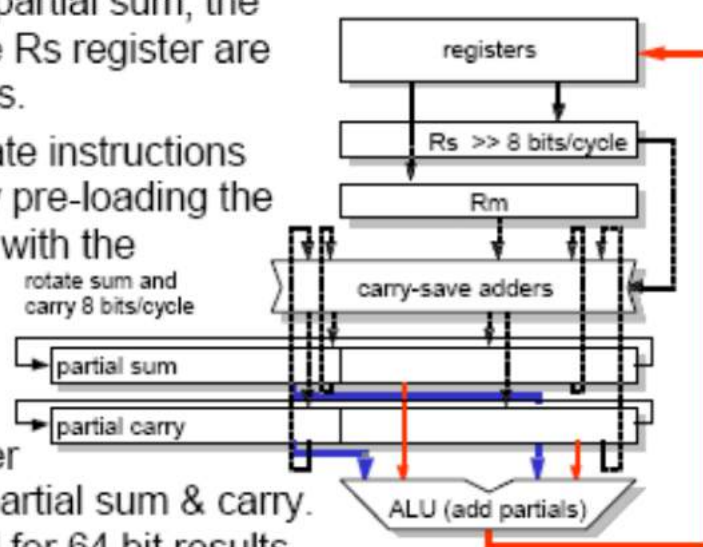
!!! Very challenging

## ARM high-speed multiplier

On each cycle, the partial sum, the partial carry and the Rs register are right shifted by 8 bits.

Multiply & accumulate instructions are implemented by pre-loading the partial sum register with the accumulate value.

On the last cycle (shown in red) the standard ARM adder is used to add the partial sum & carry. Path in blue is used for 64 bit results.



The maximum propagation delay is a carry path of 4 gates; short enough to be completed in one clock cycle.

## Signed and unsigned products

The ARM microprocessor supports both forms of long multiplication with two different instructions; `UMULL` for unsigned integers and `SMULL` using 2's complement.

The ARM multiplier uses Booth's algorithm with a dedicated carry save adder to complete a 32 bit multiplication in 5 cycles. It can complete execution in less than 5 cycles if multiplier has leading zeros.

...

# Lecture 8

---

Processor Modes & Thumb Code

## Processor Modes

---

The ARM processor has 7 'processor modes'

- Usermode
- FIQ mode
- IRQ mode
- Supervisor
- Abort
- Undef
- System

Supervisor mode is entered on reset or when a software interrupt (SWI) instruction is executed.

Abort mode is used when a memory access violation occurs, either on an instruction fetch or on a data read/write.

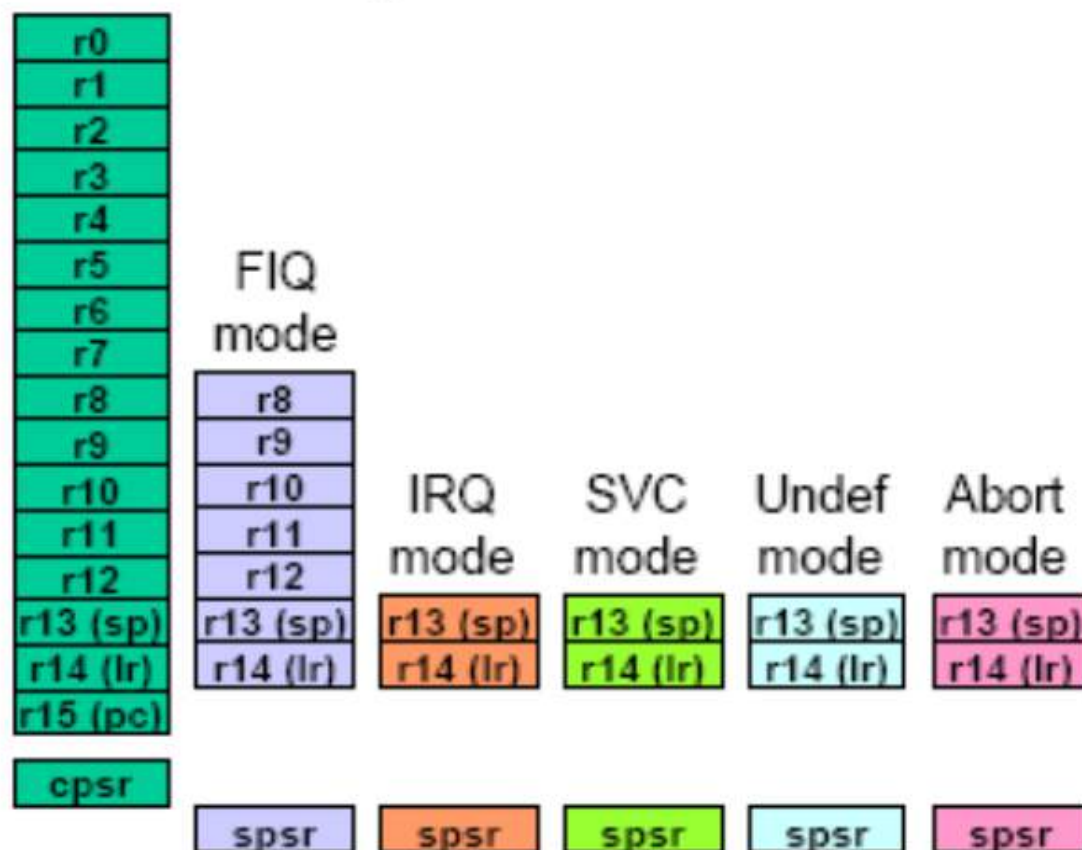
Undef mode is entered when the instruction decoder encounters machine code for which there is no instruction.

System mode is a privileged mode used for the operating system.

## Different registers in each mode

# User mode registers

i



## Current program status register cpsr

31	28				7	6	5	4	3	2	1	0
N	Z	C	V	unused	I	F	T	mode				
								User:	10000			
								FIQ:	10001			
								IRQ:	10010			
								SVC:	10011			
								Abort:	10111			
								Undef:	11011			
								System:	11111			

When the processor changes mode the 'old' contents of the cpsr are copied into the saved program status register (spsr) for the new mode.

There are five spsr registers, one for each mode except User mode and System mode.

## Exception vectors

When an interrupt occurs the program counter (r15) is reloaded with the exception vector value.

We have already covered the exception vectors for IRQ (0x00000018) and FIQ (0x0000001C).

When the power is first connected to the processor, a reset occurs and the program counter is loaded with the exception vector value `0x00000000` (usually an unconditional branch instruction).

## Vector Table

Exception	Mode	vector address
Reset	SVC	0x00000000
Undefined instruction	Undef	0x00000004
Software interrupt (SWI)	SVC	0x00000008
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C
Data abort (data access memory fault)	Abort	0x00000010
IRQ (normal interrupts)	IRQ	0x00000018
FIQ (fast interrupts)	FIQ	0x0000001C

## Action upon an exception

1. The `cpsr` is copied into the relevant `spsr`.
2. The processor switches mode – the mode bits in the `cpsr` are changed as appropriate.
3. If appropriate, interrupts are disabled by setting the relevant bit or bits in the `cpsr`, e.g. an FIQ disables IRQ.
4. The return address is stored in the relevant link register.
5. The program counter is reloaded with the relevant vector address.

## Thumb instruction set

Thumb instructions are 16 bit compressed ARM instructions.

```
AND r6, r5 0x402E
```

the value in register r5 with the value in register r6 and leave the result in register r6.

```
ADD r1, #0x36 0x3136
```

add 54 to value in register r1 and put the sum in register r1.

## Restrictions

In order to fit Thumb instructions into 16 bits, many of the features of ARM instructions are lost.

- Most Thumb instructions are not conditionally executable
- only branches can be conditional.
- Only `r0` to `r7` are used with most Thumb instructions.
- Most Thumb instructions use one of the source registers as the destination register.
- Immediate values can not be rotated.

- There is no option on the setting of flags – most Thumb instructions set the flags.

## Thumb de-compressor

The ARM instruction decoder includes a combinational logic circuit known as the Thumb de-compressor.

When the processor is executing Thumb code (T bit set in the `cpsr`) then each 32 bit word is split into two half words.

One of the 16 bit codes is 'mapped' into the equivalent 32 bit ARM code (e.g. `0x402E` converts to `0xE0166005`) and the processor decodes and executes that instruction.

On the next clock cycle the de-compressor converts the other half word into the equivalent ARM code which can then be decoded and executed.

## Thumb – why?

Thumb code uses approx. 40% more instructions for a given task than ARM code.

As a result Thumb code occupies about 70% of the memory space used by ARM – it has a better 'code density' – and it uses 30% less external memory power as a result.

## Thumb v. ARM

ARM code is 40% faster than Thumb code if instructions are fetched on a 32 bit bus, so in a system where performance is paramount, ARM code and a 32 bit memory system are used.

However in a 16 bit memory system, Thumb code is 45% faster than ARM code. In a system where memory cost and power consumption are important then a 16 bit memory system and Thumb code would be the better choice.

Most systems use a bit of each; **ARM code for the critical routines and Thumb code for background tasks.**

