Lecture 6

Memory

Memory

- Memory can be defined as any device that can store information.
- In the context of microprocessors, memory is a device that can store information in a binary format.
- Historically computer memory included punch cards, magnetic tape, magnetic core - all now obsolete.
- Most memory is now in the form of semiconductor integrated circuits, hard disks, CD-ROMs, *etc*.

Types of Silicon Memory

- Silicon IC memory can be classified as either read-only or read-write.
- Read-only memory (or ROM) is 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 (which is normally called RAM) is used for an application program on a desktop computer and for temporary data.

Types of ROM

- There are several generations of silicon 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.

Types of ROM

- Flash Erasable PROM (FEPROM 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. The data stored in ROM will remain almost indefinitely e.g. many tens of years.
- In contrast the contents of RAM memory will be lost almost as soon as power is switched off - RAM is volatile.

RAM

- The acronym RAM stands for Random Access Memory
  - this means that 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.
- Although most silicon memory including ROM is 'random access' in the above sense the term RAM has become synonymous with silicon read-write memory.
- There are two types of RAM - static and 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 to store the same amount of data
  - hence SRAM is more expensive than DRAM for the same amount of storage.

Memory: fast or big?

- In general a bigger memory is a slower memory e.g.
  - A hard disk can store a few gigabytes but has an access time of tens of milliseconds.
- The main memory of a microprocessor is typically DRAM with an access time of around 100 nanoseconds and DRAM chips hold up to 32 megabytes of data.
- Memory can be placed on the same IC as the processor ('on chip' memory) and may have an access time of around 10 nanoseconds but only hold up to 32 kilobytes of data.

Fast and big!

- However the microprocessor may be executing one instruction every 5 nanoseconds so how can a processor be connected to a very large memory with a very fast access time?
- The key is to use a memory cache - this is a small fast on chip memory that holds the most recently accessed data from the main memory.
- If main memory can be compared to a telephone directory then the cache could be compared to a personal address book.

Cache memory

- When data is read from, or written to, main memory a copy of the data is saved in the memory cache (along with the main memory address).
- When a subsequent read occurs the cache can be checked to see if the required data is already there.
- If it is (a cache ‘hit’) then it can be supplied immediately.
- If not (a cache ‘miss’) then it must be fetched from main memory.

Cache memory

- The operation of a cache relies on two features of microprocessor programs;
  - temporal and spatial locality.
- Temporal locality occurs because data accessed once is likely to be accessed again soon e.g.
  - an instruction in a program loop.
- 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. (Typically the data stored in the cache may be 16 adjacent bytes.)

Cache memory

- The performance of a cache is measured by the ‘hit rate’
  - that is the fraction of memory accesses satisfied by the cache.
- The ‘hit rate’ should be over 90% to achieve the best results with modern microprocessors.
- Sometimes there may be two levels of cache
  - a ‘primary cache’ that is on chip and
  - an off chip ‘secondary cache’.
- Also there may be separate caches for instructions and data
  - the ‘modified Harvard’ architecture.

Cache operation / behavior

- The ratio of number of hits to total accesses is the **HIT RATIO**, ***h***.
- The ratio of number of misses to total accesses is the **MISS RATIO**, ***m***.

  ***h*=(1-*m*)** and ***m*=(1-*h*)**
- HIT ratios over 0.95 (95% hits) can be obtained by good cache design.
- ***h*** will depend upon the program running.

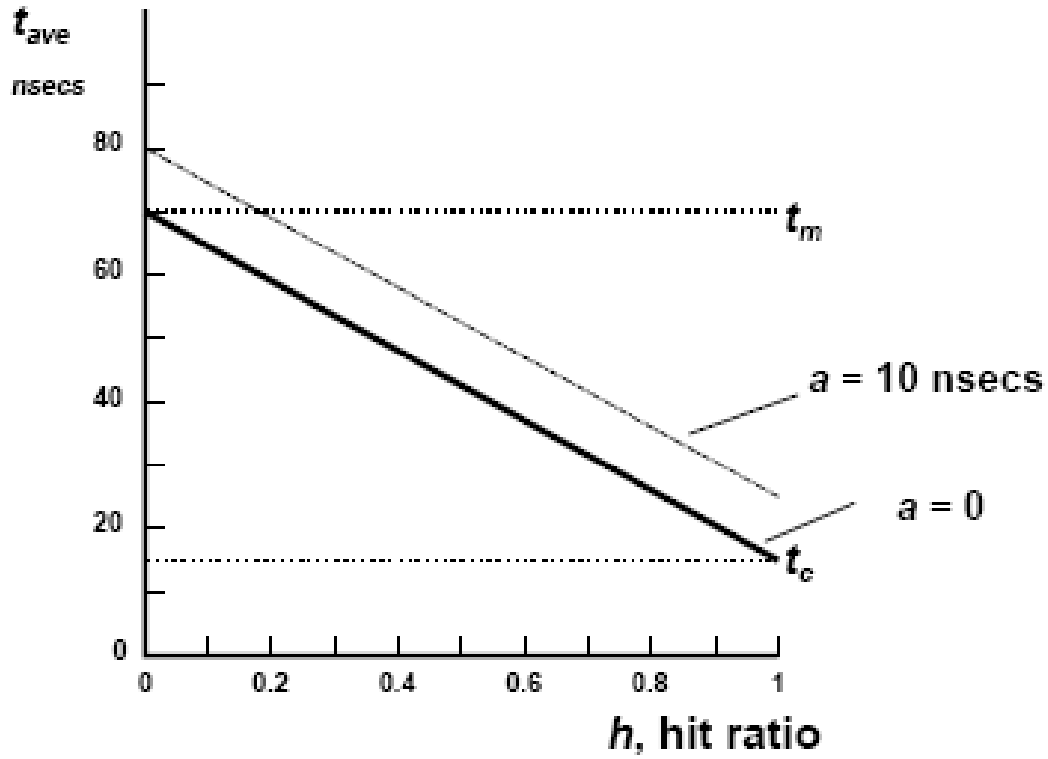
Mean access time

- Simple approach (ignoring cache controller overheads) analysis suggests for many accesses with a cache access time of t_c, and a main memory access time of t_m the mean access time will be

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

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

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



t_c = 15 nsecs, t_m = 70 nsecs

Cache design problems

- How to detect if a real address contents are already in cache.
- How to handle write operations.
- How to perform block transfers.
- Method of selecting which block to remove.

Multiple levels of cache

- Very complex multiple cache systems are used with high performance CPUs, these have multiple caches operating at different levels.
- Size/performance of memory for small to medium size CPU systems such as ARM (around 2004-5) .

	Typical Size	Access time
CPU registers	64 to 256 bytes	1 cycle
Level 1 cache	16k to 64k bytes	1 to 2 cycles
Level 2 cache	128k to 1M bytes	8 cycles
Main memory	256M to Gbytes	16 cycles
Back-up	Very large	Block transfer

Cache control systems

- Many different designs - main problem – the controller must detect if the location required is in the cache (or not) extremely quickly - a few nanoseconds.
- If in the cache it must also find the location in cache.
- Two main designs:
  - associative cache
  - direct mapped cache.
- Reminder
  - block = several bytes of main memory copied to one cache line.

Associative cache

- 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).

Associative cache organization

Cache line number ↓	Valid bit	Block number	Block contents
0			
1			
2			
3	1	10110010	010010100100100011110 1….
last			
No. of bits→	1	$Y - a$	8×2^a

8 bits per byte

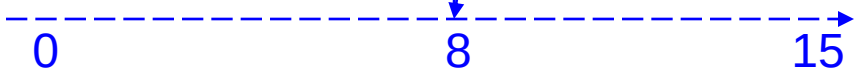
number of bytes

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

Fully associative cache – example

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

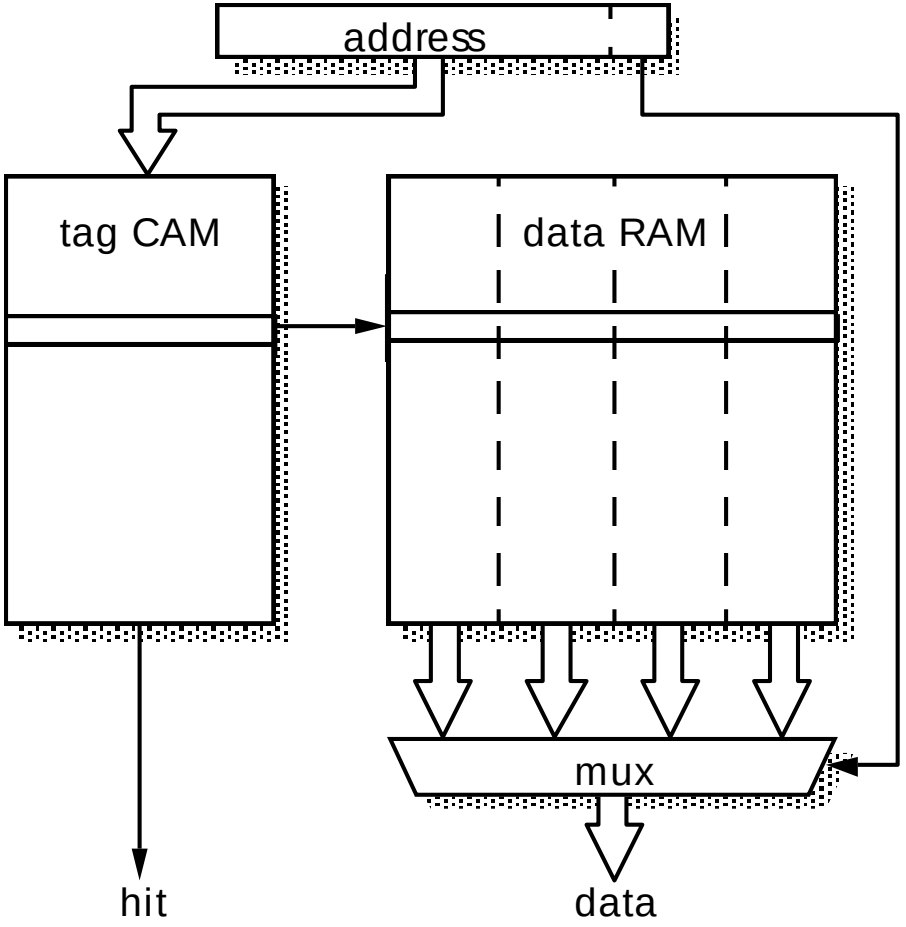


Fully associative cache organization

The content addressed memory (CAM) adds to the complexity of the total circuit.

A CAM cell is the same as a RAM cell but with a comparator.

The CAM based tag memory performs parallel comparisons to identify a hit on any of the cache lines.



Action of associative cache controller

on every CPU memory access:

1. Use the required block address.
2. Simultaneously compare every block number in the cache with that of the required location – very expensive in logic gates.
3. If address match is found check valid bit.
4. If valid bit indicates cache holds copy use cache location.
5. If valid bit indicates cache does not hold copy perform normal memory access AND copy main memory contents to a cache location (controller has to select which one to use).

Direct-mapped cache

- Avoids cache search by relating cache line number to the main memory address.
- There are $2^{\mathbf{M}}$ 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^{a} 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.**

Direct-mapped cache organization

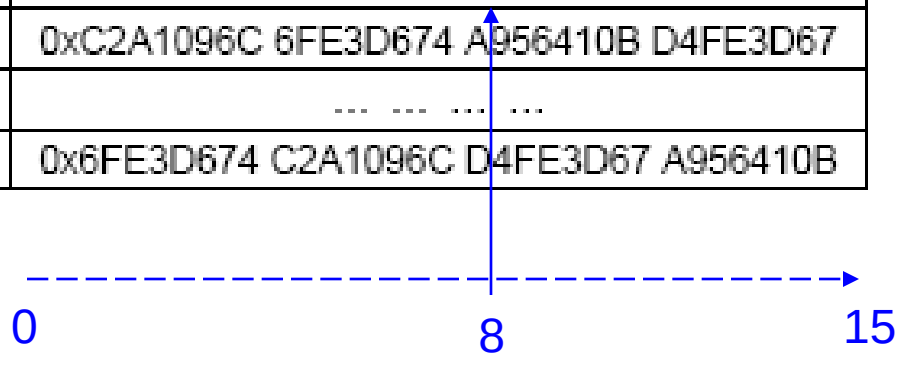
Cache line number ↓	Valid bit	Tag field	Block contents
0			
1			
2			
3	1	01011010	010010100100100011111010…
2^M-1			
No. of bits→	1	Y – M – a	8×2^a

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……..

Direct-mapped cache – example

In this example the cache line length is 16 bytes and there are 256 lines. The data byte at main memory address 0x37B6A038 is 0xD4 – highlighted.

Cache line no.	Tag field	Block contents
0x00	0x07D3A	0xC2A1096C 6FE3D674 A956410B D4FE3D67
0x01	0x9AB63	0x6FE3D674 C2A1096C D4FE3D67 A956410B
0x02	0xFF4E5	0xC2A1096C 6FE3D674 A956410B D4FE3D67
0x03	0x37B6A	0x6FE3D674 C2A1096C D4FE3D67 A956410B
0x04	0xF31A0	0xC2A1096C 6FE3D674 A956410B D4FE3D67
…	…	… … … …
0xFF	0x54B09	0x6FE3D674 C2A1096C D4FE3D67 A956410B



Direct-mapped cache organization

In the example the cache holds 4Kbytes (256 x 16) of data when full.

The tag field occupies (256 x 2.5) bytes of memory.

All main memory addresses 0xXXXXX03X (where X can be any value from 0 to F) are related to cache line number 0x03.

So different main memory addresses will contend for the same cache line. e.g. 4000403X, 5032803X

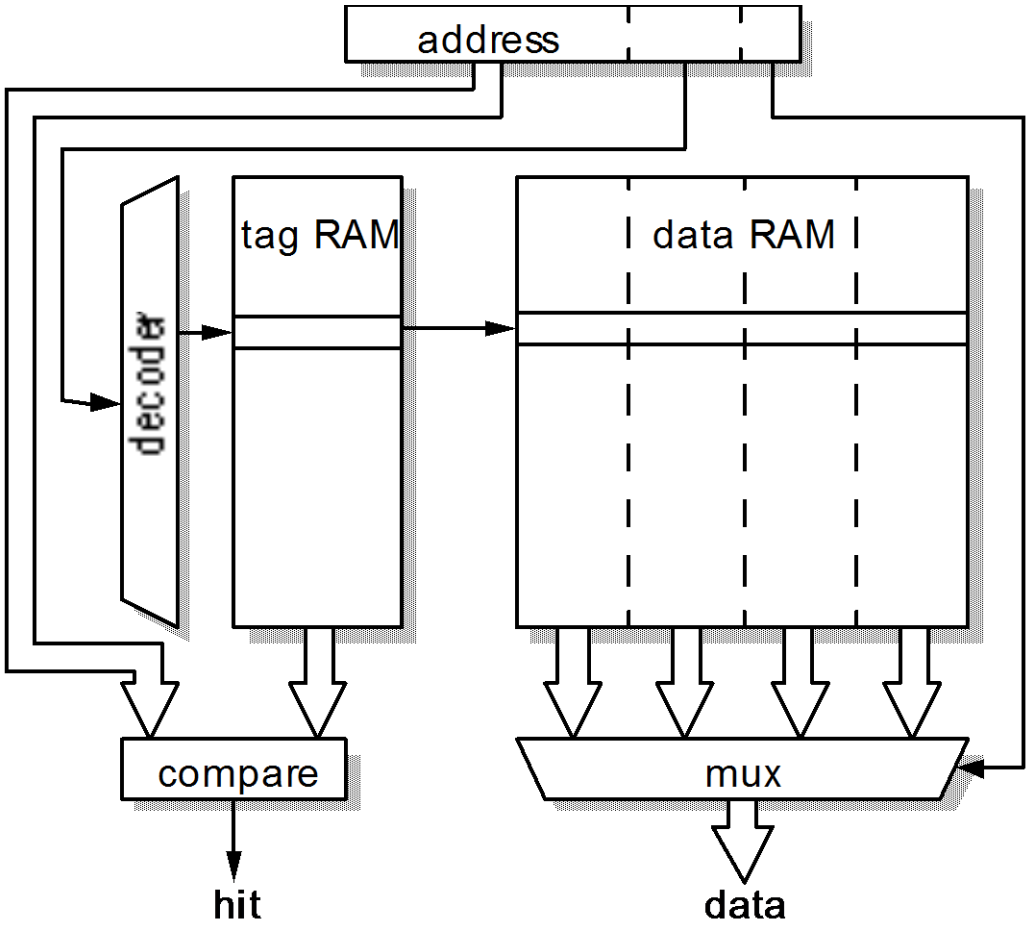
Each cache line also contains a **valid bit** to show if the line holds a valid copy of main memory contents (all set invalid at power up).

Direct-mapped cache controller

1. Use main address least significant **M** + **a** bits to access the cache line.
2. Check valid bit of line and compare tag field bits with most significant bits of main address.
3. If valid AND tag address matches, use cache location.
4. If cache does not hold copy perform normal memory access and copy main memory contents to the cache location set by the main address least significant bits (controller no longer has to select location).

Many modifications and mixed systems exist.

Direct-mapped cache organization



Design problems for cache systems

1. How to transfer a block quickly when a block is more than one word.
2. Deciding which entry to remove when cache is full and new entry required – applies only to associative cache.
3. Decision must be *very fast*, possibilities are:
   a) entry which has not been accessed for the longest time,
   b) entry which has been in longest,
   c) replace at random (easiest) – *this reduces the hit ratio, can be acceptable if cache is reasonably big*

Write access problem

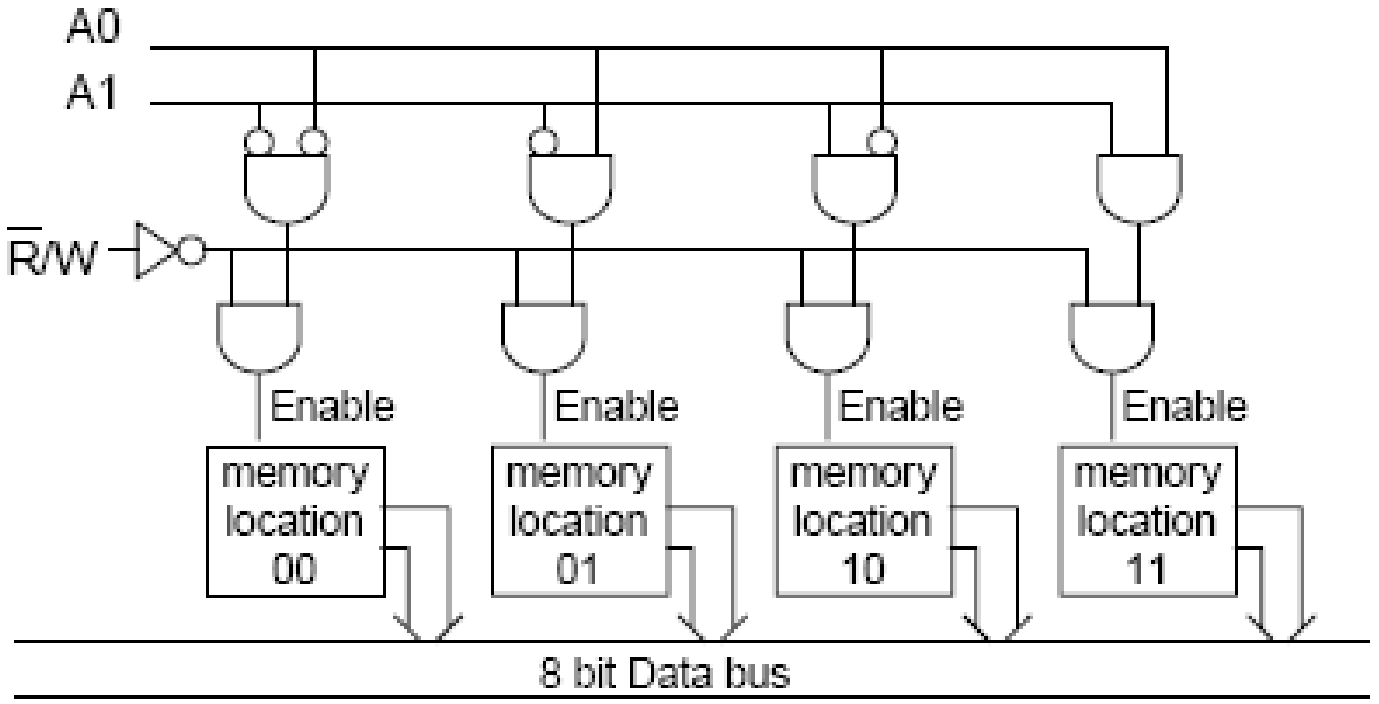
Write access problem - many arguments as to best solution.

- Problem - on a hit (write) the system will change cache contents but main memory will retain the old value. The main memory must match the cache value if the cache entry is removed.
- Two basic solutions – write through and copy back.

Write through / copy back

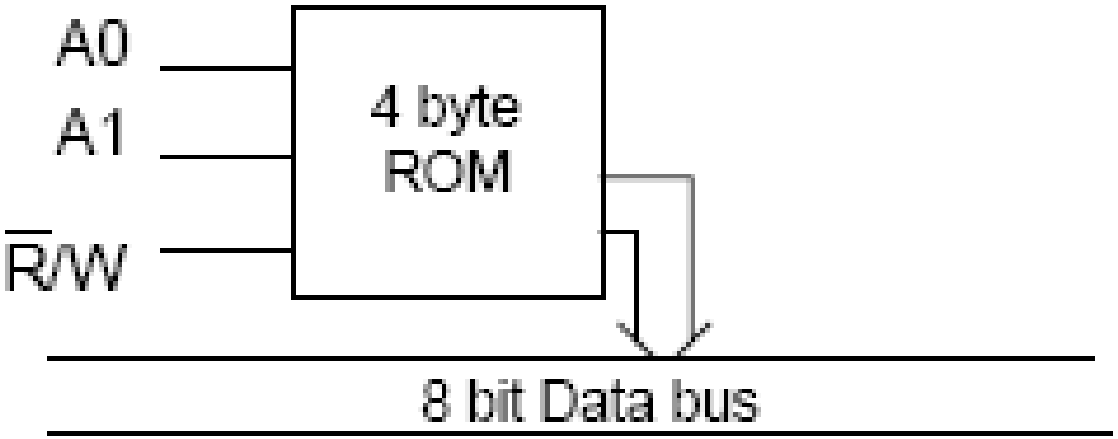
- **Write through** - every cache write, simultaneously writes to main memory. *Slows system to an extent but read more common than write.*
- **Copy back** - cache line has an extra bit that is cleared on loading, if a write is performed, it is set.
- When the cache entry is replaced, the bit is checked and, if set, the contents are written to main memory.
- 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



Simple ROM addressing

- So memory location 00 is ‘enabled’ and connected to the data bus only when R*/W = 0, A0 = 0 and A1 = 0.
- The previous circuit can be represented by a single block as follows:



Tri-state gates

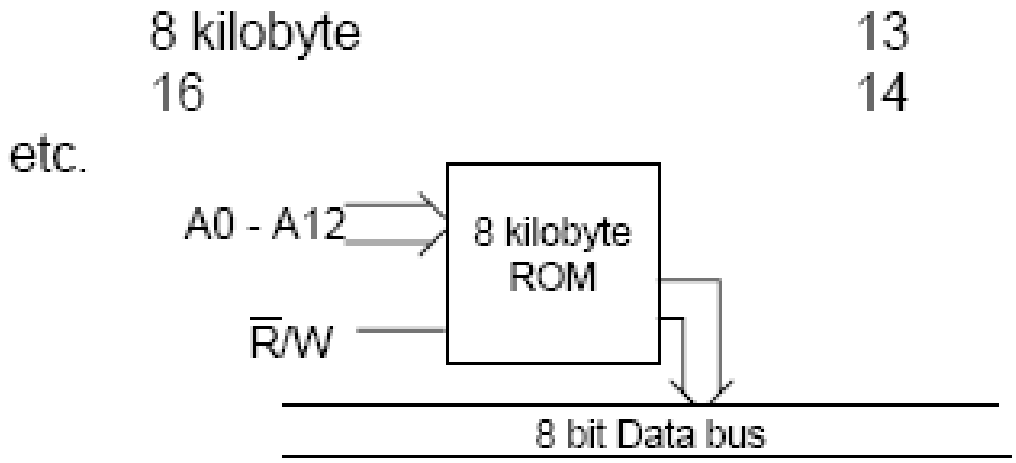
- Only one memory location should be connected to the bus at one time – one device can not drive a logic 0 onto a bus line at the same time as another device drives a logic 1 onto the same line.
- This is achieved using ‘tri-state gates’ which have an ‘enable’ input – when not enabled the output acts like an ‘open circuit’ which is referred to as ‘high impedance’.

Tri-state gates

enable	input	output
0	0	high-impedance
0	1	high-impedance
1	0	0
1	1	1

Simple ROM addressing

- Bigger memories would have similar connections but there would be more address lines.
- A 2^N byte memory would need N address lines:
- E.g. a 4 kilobyte memory would need 12 address lines.



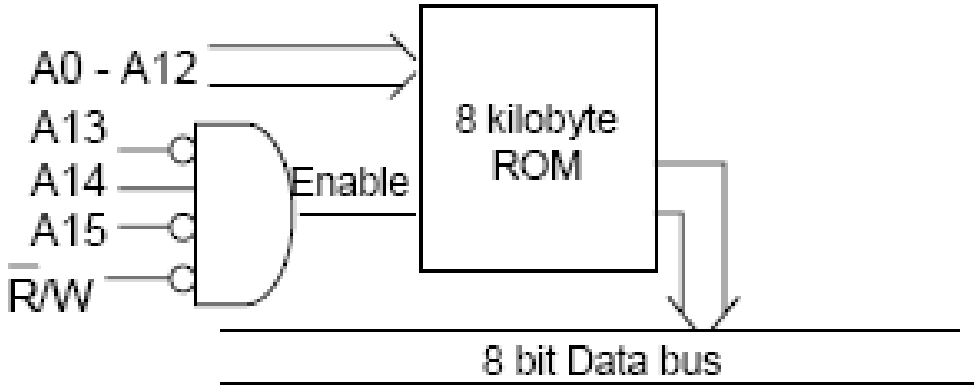
Which address?

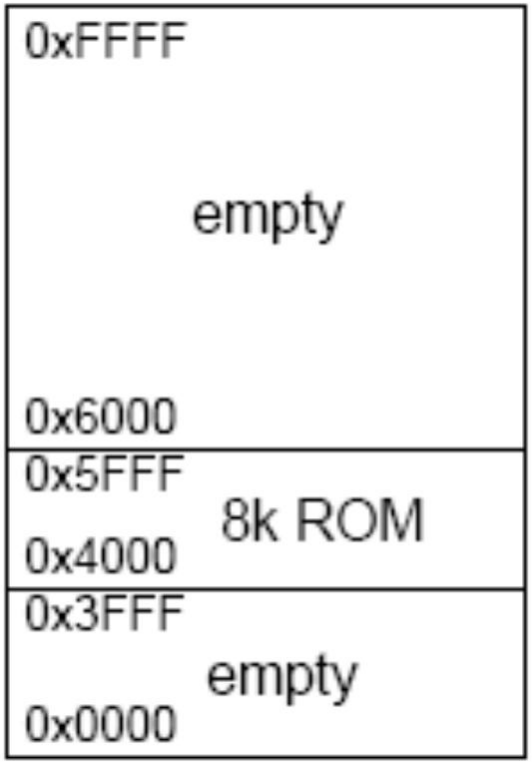
- If an 8 kilobyte memory chip is connected to a processor with a 16 bit address bus whereabouts is the memory located within in the 64 kilobyte address space?
- The lowest N address lines are connected to the 2^N byte memory; in this case A0 to A12. The other address lines, A13 to A15, must be decoded e.g. if the memory range is from 0x4000 to 0x5FFF or in binary:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

8 kilobyte ROM addressing

- The memory is only enabled when A13 = 0, A14 = 1, A15 = 0 and R*/W = 0.

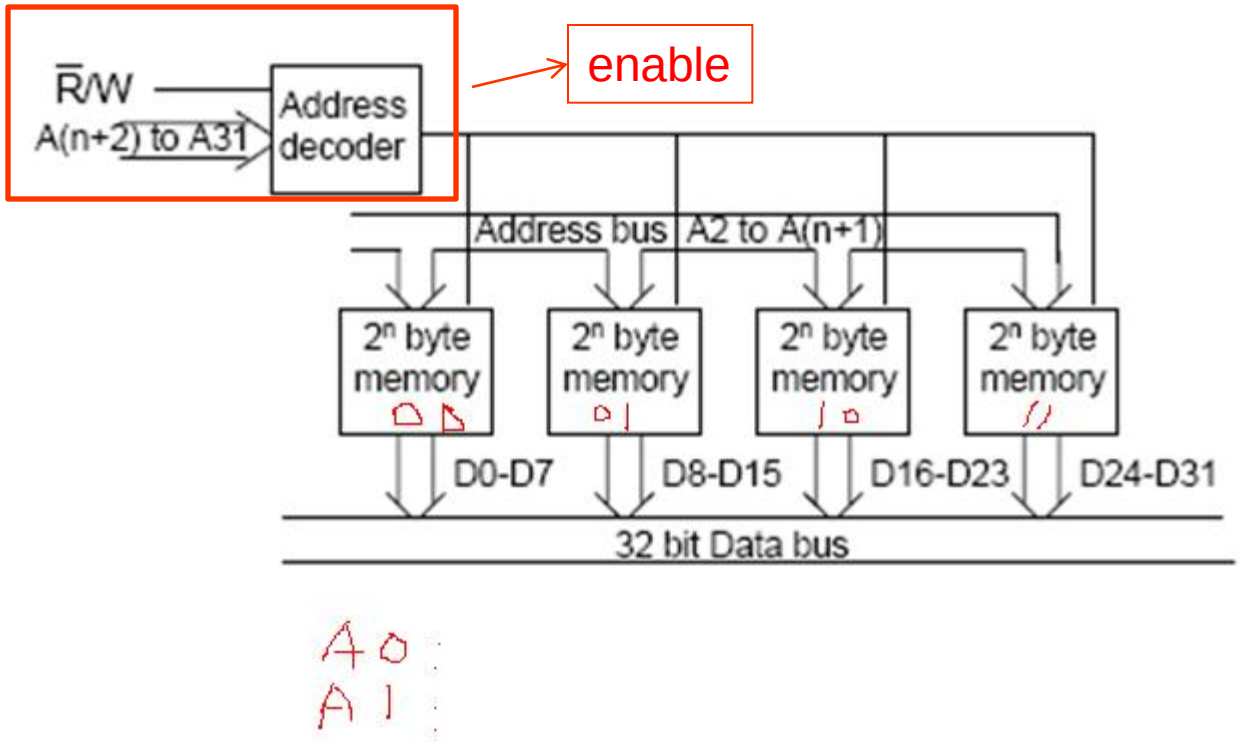




- If the address bus is 32 bits wide then all the address lines not connected to the memory chip, A13 to A31, are decoded.

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

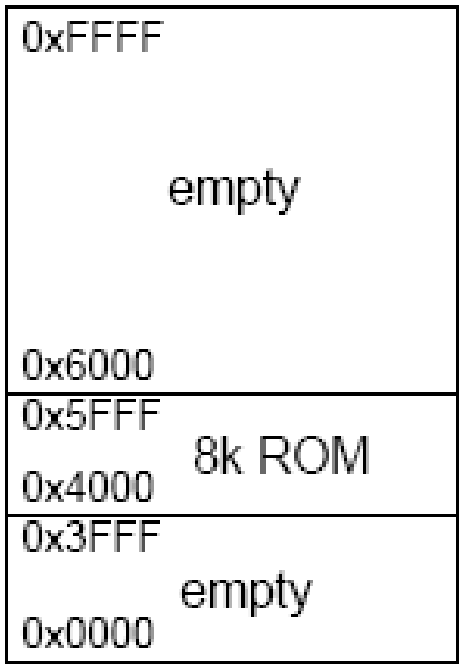


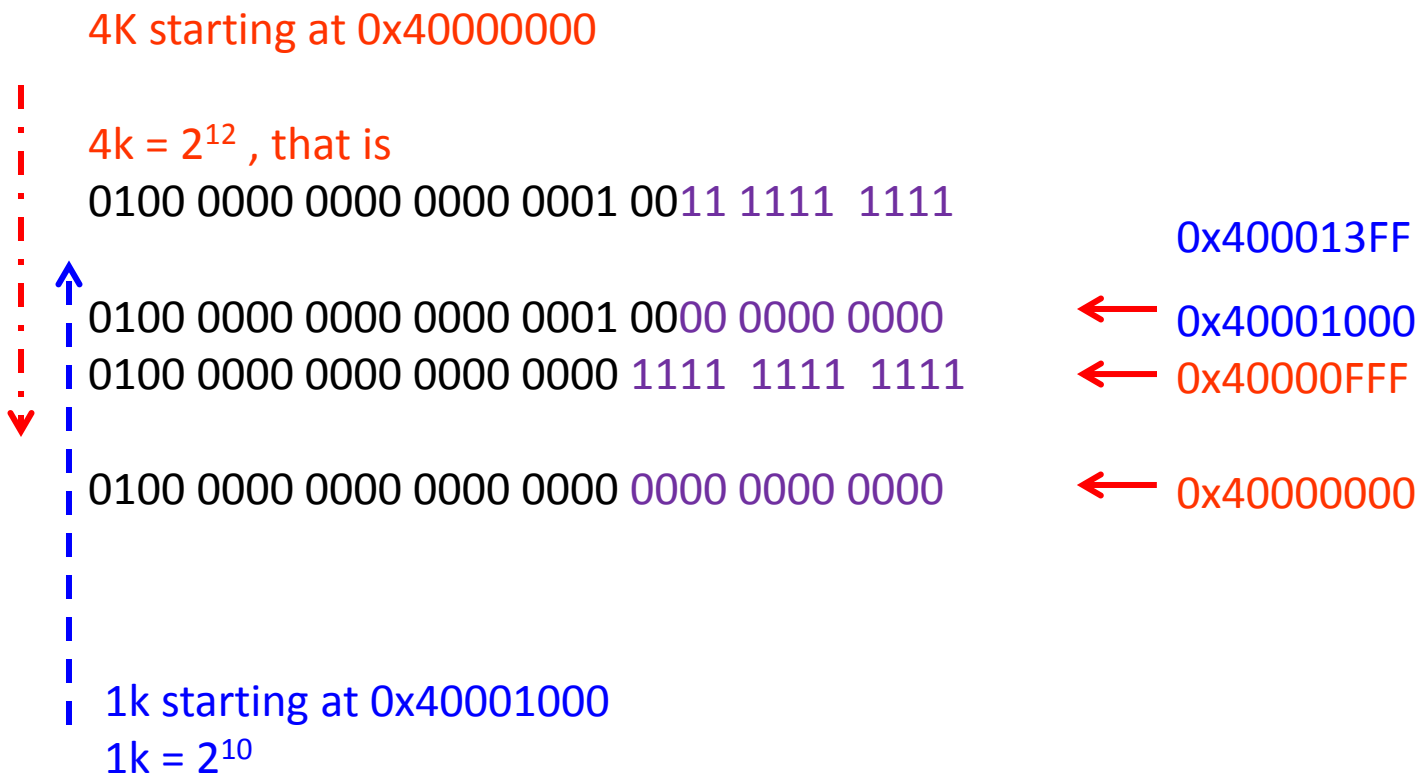
$A_0 – A_{n+1}$: memory size

$A_{n+2} – A_{31}$: location in the memory map

Memory maps

- The location of a memory chip within the complete addressable space can be represented by a memory map.
- E.g. for the 8 kilobyte ROM memory between addresses 0x4000 and 0x5FFF in a 64 kilobyte addressable space:




4K starting at 0x40000000
4k = 2^12 , that is
0100 0000 0000 0000 0001 0011 1111 1111
0x400013FF
0100 0000 0000 0000 0001 0000 0000 0000
0x40001000
0100 0000 0000 0000 0000 1111 1111 1111
0x40000FFF
0100 0000 0000 0000 0000 0000 0000 0000
0x40000000
1k starting at 0x40001000
1k = 2^10

Input and output

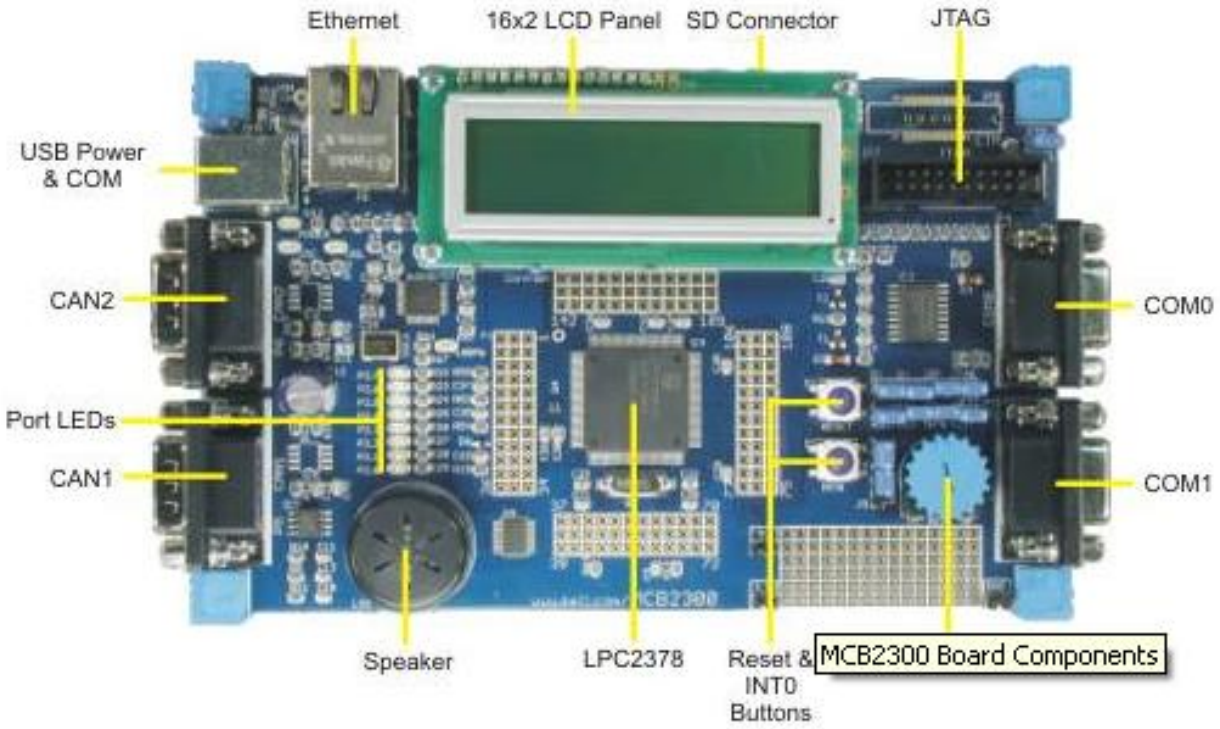
- All microprocessor systems communicate with the real world in some way - e.g.
  - a video game takes an input from a joystick and returns an output to a video screen,
  - an automatic pilot takes an input from an altimeter and returns an output to control the position of flaps/rudders.
- Input to and 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.

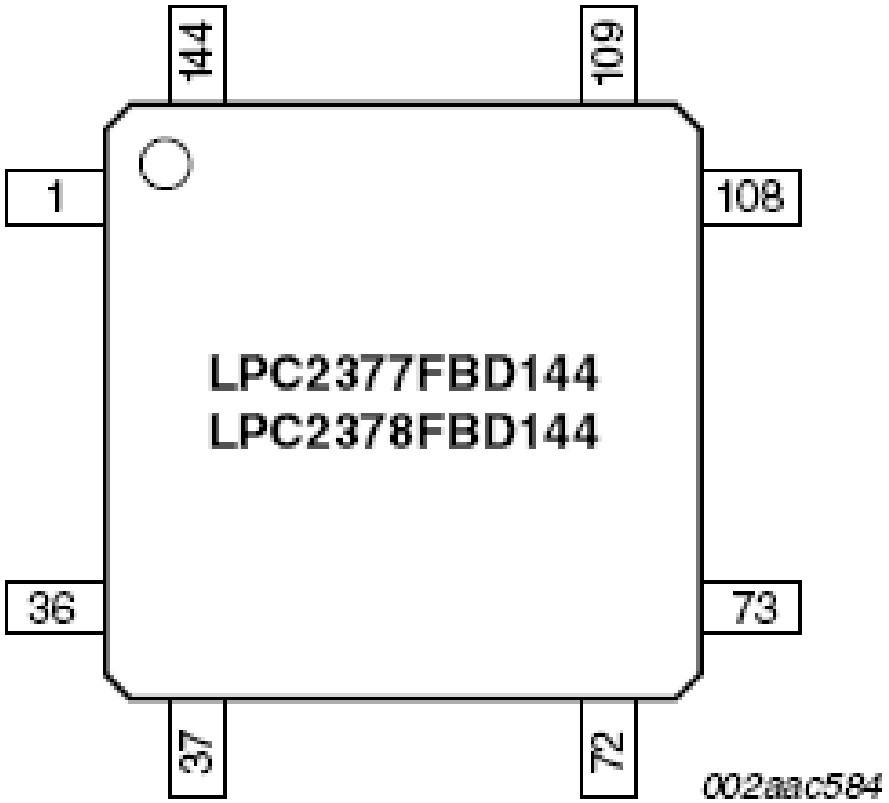
Memory mapped I/O

- Many microprocessors, include the ARM, use ‘memory mapped’ input and output.
- The ARM7 has a 32 bit address and can address 2^{32} bytes or 4 gigabytes of memory.
- 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.

Memory mapped I/O

- For example the addresses 0x3FFFC058 and 0x3FFFC05C are used on the ARM7 development system (LPC2378) for input and output for port 2, respectively.
- The 8 output LEDs are connected to bit 0 to bit 7 of port 2.
- When the address 0x3FFFC058 bit 2 is set the value 1, the third LED from the bottom is light up.




144
109
1
108
LPC2377FBD144
LPC2378FBD144
36
73
37
72
002aac584

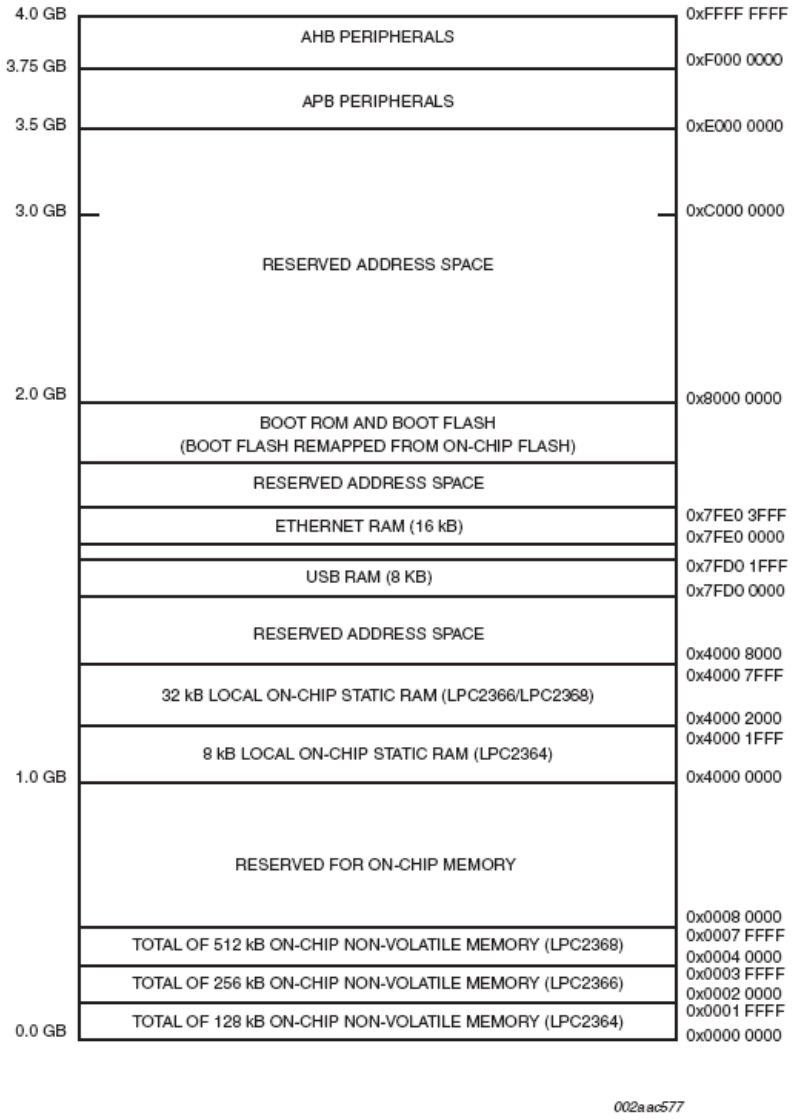
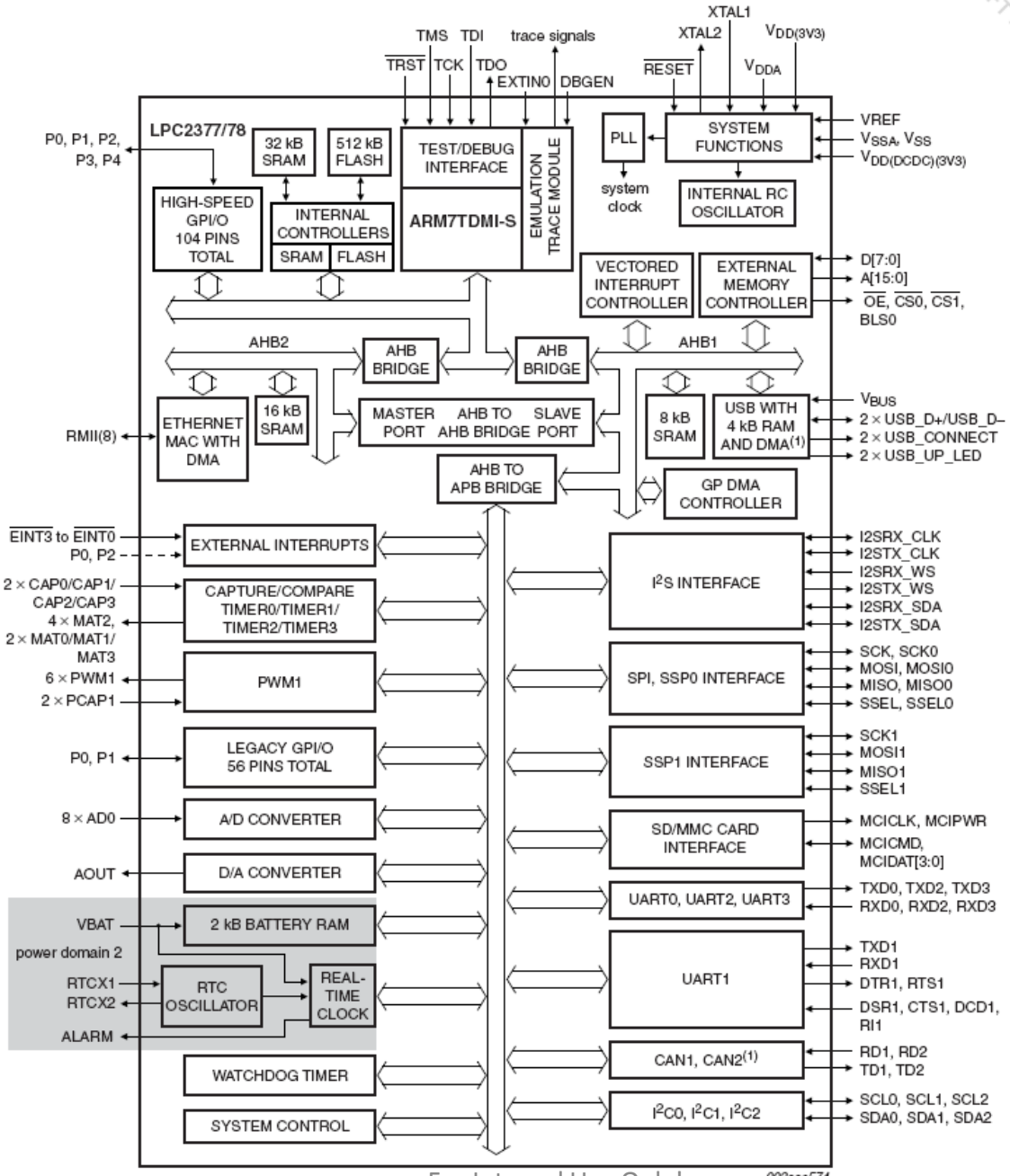


Fig 3. LPC2364/66/68 system memory map



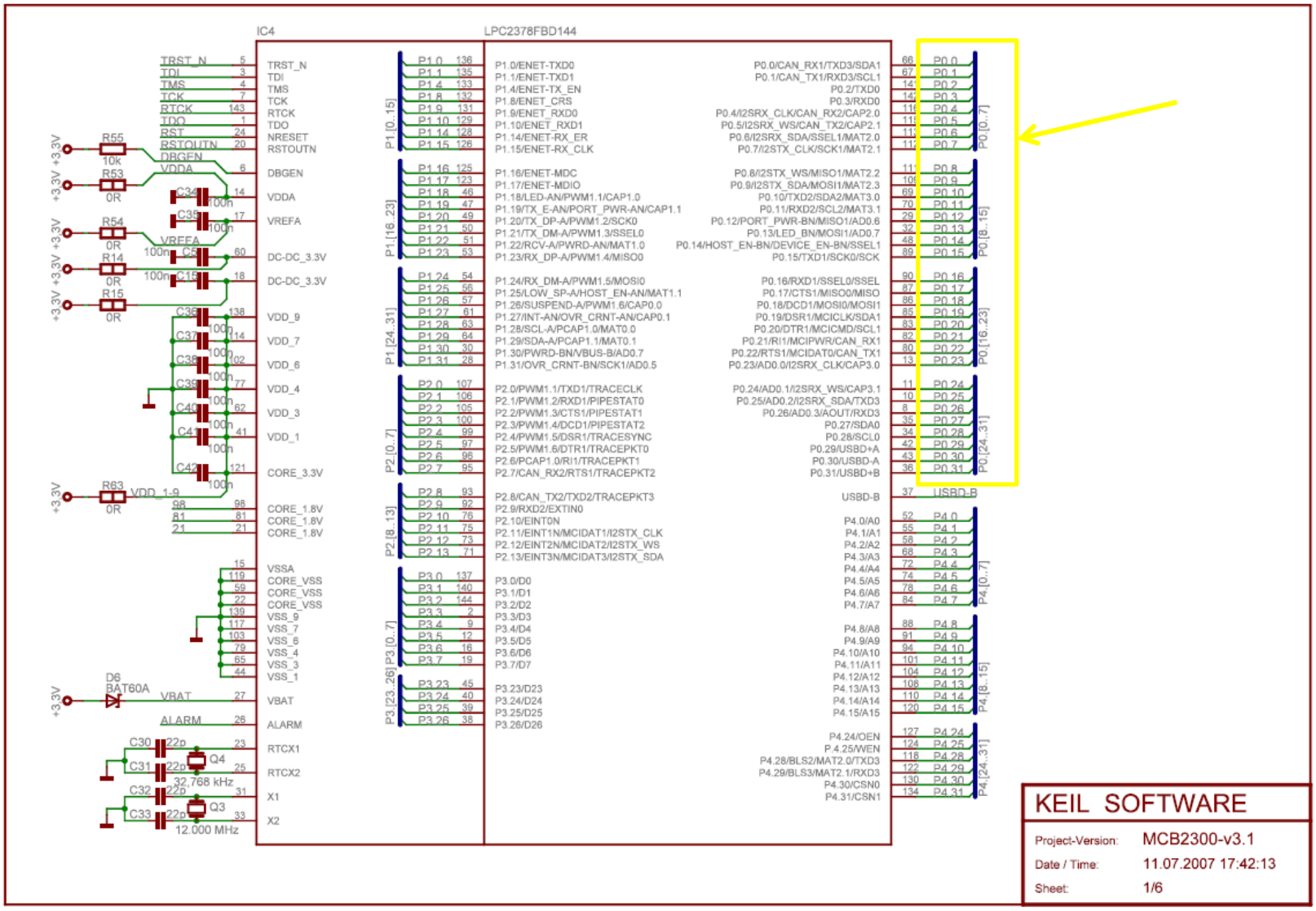

IC4
LPC2378FBD144
TRST_N
TDI
TMS
TCK
RTCK
TDO
RST
RSTOUTN
DBGEN
VDDA
VREFA
DC-DC_3.3V
VDD_9
VDD_7
VDD_6
VDD_4
VDD_3
VDD_1
CORE_3.3V
VDD_1-9
CORE_1.8V
VSSA
CORE_VSS
VSS_9
VSS_7
VSS_6
VSS_4
VSS_3
VSS_1
VBAT
ALARM
RTCX1
RTCX2
X1
X2
32.768 kHz
12.000 MHz
D6
BAT60A
P1.[0..15]
P1.[16..23]
P1.[24..31]
P2.[0..7]
P2.[8..13]
P3.[0..7]
P3.[23..26]
P0.[0..7]
P0.[8..15]
P0.[16..23]
P0.[24..31]
P4.[0..7]
P4.[8..15]
P4.[24..31]
USBD-B
KEIL SOFTWARE
Project-Version: MCB2300-v3.1
Date / Time: 11.07.2007 17:42:13
Sheet: 1/6

Table 3. LPC2300 memory usage

Address range	General use	Address range details and description	
0x0000 0000 to 0x3FFF FFFF	On-Chip NV Memory and fast I/O	0x0000 0000 - 0x0007 FFFF	Flash Memory (up to 512 kB)
		0x3FFF C000 - 0x3FFF FFFF	Fast GPIO registers
0x4000 0000 to 0x7FFF FFFF	On-Chip RAM	0x4000 0000 - 0x4000 7FFF	RAM (up to 32 kB)
		0x7FD0 0000 - 0x7FD0 1FFF	USB RAM (8 kB)
		0x7FE0 0000 - 0x7FE0 3FFF	Ethernet RAM (16 kB)
0x8000 0000 to 0xDFFF FFFF	Off-Chip Memory	Two static memory banks, 64 KB each:	
		0x8000 0000 - 0x8000 FFFF	Static memory bank 0, 64 KB
		0x8100 0000 - 0x8100 FFFF	Static memory bank 1, 64 KB
0xE000 0000 to 0xEFFF FFFF	APB Peripherals	0xE000 0000 - 0xE008 FFFF	36 peripheral blocks, 16 kB each (some unused).
		0xE01F C000 - 0xE01F FFFF	System Control Block
0xF000 0000 to 0xFFFF FFFF	AHB Peripherals	0xFFE0 0000 - 0xFFE0 3FFF	Ethernet Controller
		0xFFE0 4000 - 0xFFE0 7FFF	General Purpose DMA Controller
		0xFFE0 8000 - 0xFFE0 BFFF	External Memory Controller (EMC)
		0xFFE0 C000 - 0xFFE0 FFFF	USB Controller
		0xFFFF F000 - 0xFFFF FFFF	Vectored Interrupt Controller (VIC)

Table 4. APB peripherals and base addresses

APB Peripheral	Base Address	Peripheral Name
0	0xE000 0000	Watchdog Timer
1	0xE000 4000	Timer 0
2	0xE000 8000	Timer 1
3	0xE000 C000	UART0
4	0xE001 0000	UART1
5	0xE001 4000	Not used
6	0xE001 8000	PWM1
7	0xE001 C000	I^2C0
8	0xE002 0000	SPI
9	0xE002 4000	RTC
10	0xE002 8000	GPIO
11	0xE002 C000	Pin Connect Block
12	0xE003 0000	SSP1
13	0xE003 4000	ADC
14	0xE003 8000	CAN Acceptance Filter RAM
15	0xE003 C000	CAN Acceptance Filter Registers
16	0xE004 0000	CAN Common Registers
17	0xE004 4000	CAN Controller 1
18	0xE004 8000	CAN Controller 2
19 to 22	0xE004 C000 to 0xE005 8000	Not used
23	0xE005 C000	I^2C1
24	0xE006 0000	Not used
25	0xE006 4000	Not used
26	0xE006 8000	SSP0
27	0xE006 C000	DAC
28	0xE007 0000	Timer 2
29	0xE007 4000	Timer 3
30	0xE007 8000	UART2
31	0xE007 C000	UART3
32	0xE008 0000	I^2C2
33	0xE008 4000	Battery RAM
34	0xE008 8000	I^2S
35	0xE008 C000	SD/MMC Card Interface
36 to 126	0xE009 0000 to 0xE01F BFFF	Not used
127	0xE01F C000	System Control Block

Table 107. GPIO register map (legacy APB accessible registers)

Generic Name	Description	Access	Reset value[1]	PORTn Register Address & Name
IOPIN	GPIO Port Pin value register. The current state of the GPIO configured port pins can always be read from this register, regardless of pin direction. By writing to this register port's pins will be set to the desired level instantaneously.	R/W	NA	IO0PIN - 0xE002 8000 IO1PIN - 0xE002 8010
IOSET	GPIO Port Output Set register. This register controls the state of output pins in conjunction with the IOCLR register. Writing ones produces highs at the corresponding port pins. Writing zeroes has no effect.	R/W	0x0	IO0SET - 0xE002 8004 IO1SET - 0xE002 8014
IODIR	GPIO Port Direction control register. This register individually controls the direction of each port pin.	R/W	0x0	IO0DIR - 0xE002 8008 IO1DIR - 0xE002 8018
IOCLR	GPIO Port Output Clear register. This register controls the state of output pins. Writing ones produces lows at the corresponding port pins and clears the corresponding bits in the IOSET register. Writing zeroes has no effect.	WO	0x0	IO0CLR - 0xE002 800C IO1CLR - 0xE002 801C

[1] Reset value reflects the data stored in used bits only. It does not include reserved bits content.

Definitions for GPIO

```
/* General Purpose Input/Output (GPIO) */
#define GPIO_BASE_ADDR                0xE0028000
#define IOPIN0      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x00))
#define IOSET0      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x04))
#define IODIR0      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x08))
#define IOCLR0      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x0C))
#define IOPIN1      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x10))
#define IOSET1      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x14))
#define IODIR1      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x18))
#define IOCLR1      (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x1C))
```

Table 109. GPIO interrupt register map

Generic Name	Description	Access	Reset value[1]	PORTn Register Address & Name
IntEnR	GPIO Interrupt Enable for Rising edge.	R/W	0x0	IO0IntEnR - 0xE002 8090 IO2IntEnR - 0xE002 80B0
IntEnF	GPIO Interrupt Enable for Falling edge.	R/W	0x0	IO0IntEnR - 0xE002 8094 IO2IntEnR - 0xE002 80B4
IntStatR	GPIO Interrupt Status for Rising edge.	RO	0x0	IO0IntStatR - 0xE002 8084 IO2IntStatR - 0xE002 80A4
IntStatF	GPIO Interrupt Status for Falling edge.	RO	0x0	IO0IntStatF - 0xE002 8088 IO2IntStatF - 0xE002 80A8
IntClr	GPIO Interrupt Clear.	WO	0x0	IO0IntClr - 0xE002 808C IO2IntClr - 0xE002 80AC
IntStatus	GPIO overall Interrupt Status.	RO	0x00	IOIntStatus - 0xE002 8080

[1] Reset value reflects the data stored in used bits only. It does not include reserved bits content.

Definitions for GPIO Interrupt Registers

- /* GPIO Interrupt Registers */
- #define IO0_INT_EN_R (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x90))
- #define IO0_INT_EN_F (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x94))
- #define IO0_INT_STAT_R (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x84))
- #define IO0_INT_STAT_F (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x88))
- #define IO0_INT_CLR (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x8C))

- #define IO2_INT_EN_R (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0xB0))
- #define IO2_INT_EN_F (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0xB4))
- #define IO2_INT_STAT_R (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0xA4))
- #define IO2_INT_STAT_F (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0xA8))
- #define IO2_INT_CLR (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0xAC))

- #define IO_INT_STAT (*(volatile unsigned long *)(GPIO_BASE_ADDR + 0x80))

Table 108. GPIO register map (local bus accessible registers - enhanced GPIO features)

Generic Name	Description	Access	Reset value[1]	PORTn Register Address & Name
FIODIR	Fast GPIO Port Direction control register. This register individually controls the direction of each port pin.	R/W	0x0	FIO0DIR - 0x3FFF C000 FIO1DIR - 0x3FFF C020 FIO2DIR - 0x3FFF C040 FIO2DIR - 0x3FFF C060 FIO2DIR - 0x3FFF C080
FIOMASK	Fast Mask register for port. Writes, sets, clears, and reads to port (done via writes to FIOPIN, FIOSET, and FIOCLR, and reads of FIOPIN) alter or return only the bits enabled by zeros in this register.	R/W	0x0	FIO0MASK - 0x3FFF C010 FIO1MASK - 0x3FFF C030 FIO2MASK - 0x3FFF C050 FIO3MASK - 0x3FFF C070 FIO4MASK - 0x3FFF C090
FIOPIN	Fast Port Pin value register using FIOMASK. The current state of digital port pins can be read from this register, regardless of pin direction or alternate function selection (as long as pins are not configured as an input to ADC). The value read is masked by ANDing with inverted FIOMASK. Writing to this register places corresponding values in all bits enabled by zeros in FIOMASK. **Important:** if a FIOPIN register is read, its bit(s) masked with 1 in the FIOMASK register will be set to 0 regardless of the physical pin state.	R/W	0x0	FIO0PIN - 0x3FFF C014 FIO1PIN - 0x3FFF C034 FIO2PIN - 0x3FFF C054 FIO3PIN - 0x3FFF C074 FIO4PIN - 0x3FFF C094
FIOSET	Fast Port Output Set register using FIOMASK. This register controls the state of output pins. Writing 1s produces highs at the corresponding port pins. Writing 0s has no effect. Reading this register returns the current contents of the port output register. Only bits enabled by 0 in FIOMASK can be altered.	R/W	0x0	FIO0SET - 0x3FFF C018 FIO1SET - 0x3FFF C038 FIO2SET - 0x3FFF C058 FIO3SET - 0x3FFF C078 FIO4SET - 0x3FFF C098
FIOCLR	Fast Port Output Clear register using FIOMASK0. This register controls the state of output pins. Writing 1s produces lows at the corresponding port pins. Writing 0s has no effect. Only bits enabled by 0 in FIOMASK0 can be altered.	WO	0x0	FIO0CLR - 0x3FFF C01C FIO1CLR - 0x3FFF C03C FIO2CLR - 0x3FFF C05C FIO3CLR - 0x3FFF C07C FIO4CLR - 0x3FFF C09C

Definitions for Fast I/O

```
/* Fast I/O setup */
#define FIO_BASE_ADDR                    0x3FFFC000
#define FIO0DIR      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x00))
#define FIO0MASK     (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x10))
#define FIO0PIN      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x14))
#define FIO0SET      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x18))
#define FIO0CLR      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x1C))

#define FIO1DIR      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x20))
#define FIO1MASK     (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x30))
#define FIO1PIN      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x34))
#define FIO1SET      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x38))
#define FIO1CLR      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x3C))

#define FIO2DIR      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x40))
#define FIO2MASK     (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x50))
#define FIO2PIN      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x54))
#define FIO2SET      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x58))
#define FIO2CLR      (*(volatile unsigned long *)(FIO_BASE_ADDR + 0x5C))
```

Example 1: sequential accesses to IOSET and IOCLR affecting the same GPIO pin/bit

State of the output configured GPIO pin is determined by writes into the pin's port IOSET and IOCLR registers. Last of these accesses to the IOSET/IOCLR register will determine the final output of a pin.

In the example code:

```
IO0DIR = 0x0000 0080 ;pin P0.7 configured as output
IO0CLR = 0x0000 0080 ;P0.7 goes LOW
IO0SET = 0x0000 0080 ;P0.7 goes HIGH
IO0CLR = 0x0000 0080 ;P0.7 goes LOW
```

pin P0.7 is configured as an output (write to IO0DIR register). After this, P0.7 output is set to low (first write to IO0CLR register). Short high pulse follows on P0.7 (write access to IO0SET), and the final write to IO0CLR register sets pin P0.7 back to low level.

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.

It is usually a good idea to associate a software module called a device driver with each of the external peripherals.

The Device Driver Philosophy

- When it comes to designing device drivers, you should always focus on one easily stated goals: hide the hardware completely;
- When you're finished, you want the device driver module to be the only piece of software in the entire system that reads or writes that particular device's control and status registers directly.
- In addition, if the device generates any interrupts, the interrupt service routine that responds to them should be an integral part of the device driver.

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.