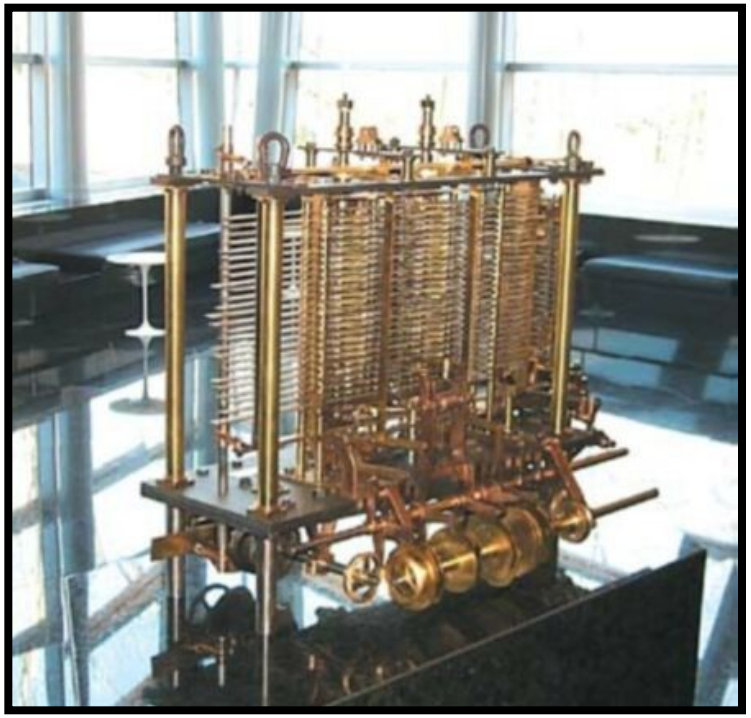
Part 3

Hardware



Part of Charles Babbage's never completed "difference engine". It would have been the world's first computer, mechanical, driven by steam

Contents

- **Lecture 11 & 12 – Input & Output, Exceptions & Interrupts**
  - Interrupt-driven vs DMA vs polled I/O
  - SWIs, exceptions & interrupts
  - ARM modes, shadow registers
- **Lecture 13 – ARM Hardware**
  - Pipelining instruction fetch and execution
  - Throughput vs Latency
  - ARM Processor organisation
- **Lecture 14 – Cache Hierarchy**
  - Adding a cache to the memory hierarchy
    - Cache purpose, hit & miss calculations
  - Direct mapped caches: basics
    - Cache lines
    - Tag, index, select fields
  - Startup issues
    - Valid bit

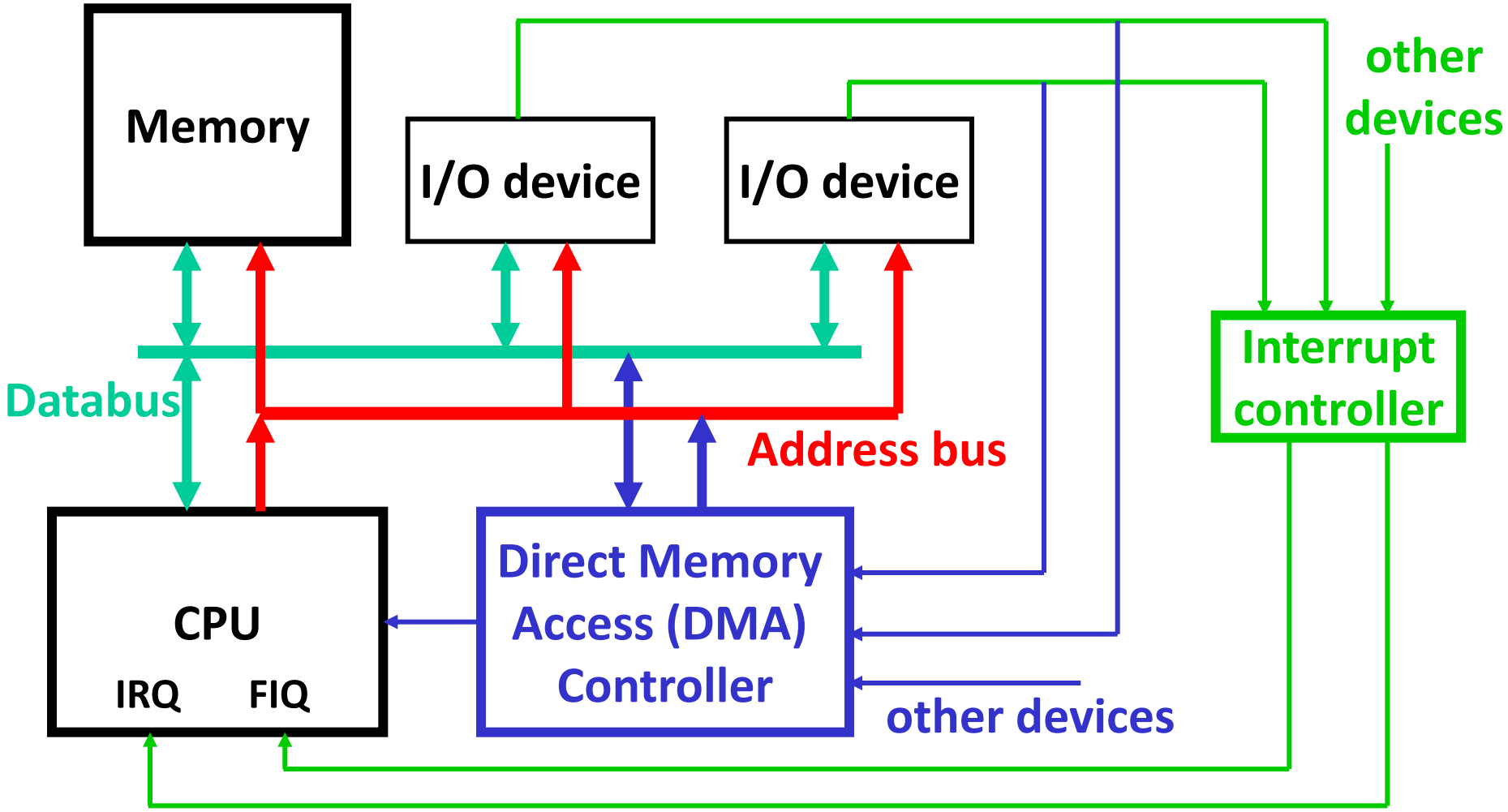
Lecture 11 & 12 I/O, Exceptions and Interrupts

"There is no useful rule without an exception" Thomas Fuller

- Handling I/O efficiently is a key requirement
- Exceptions (& interrupts) allow processors to handle events that occur which are not directly related to the user program
  - *Exceptions* – typically caused by unexpected run-time errors in user code (e.g. division by 0)
  - *Interrupts* – **special case of exception** caused by hardware conditions that require processor action – e.g. I/O service: completely independent of user code

Hardware for I/O - Overview

- I/O devices communicate via memory-mapped registers
- DMA & Interrupts are optional



Servicing an I/O device

- Typical input
  - PC serial port receives data at fixed rate (say 9600 baud) independent of CPU instructions.
  - New data byte is available 9600/10 = 960 times per second
  - CPU must read data register to input new value repeatedly, at correct time.
    - This operation is known as servicing the I/O device
- The problem
  - How can the CPU know when to service the I/O device?

Polling and Interrupt

- Both are methods to notify processor that I/O device needs attention
- **Polling**
  - simple, but slow
  - processor check status of I/O device regularly to see if it needs attention
  - similar to checking a telephone without bells!
  - responsibility of user code to check I/O device status soon enough to avoid buffer overrun.
- **Interrupt**
  - fast, but more complicated
  - processor is notified by I/O device (interrupted) when device needs attention
  - similar to a telephone with bells

Method 1: Polling

- CPU repeatedly reads status bit from serial port (bit 31 of PORT_STATUS here) to determine if new data is available
- When needed it reads data from PORT_DATA(7:0)
- Polling is simple, requires **no additional hardware**
- Requires 100% CPU use, inefficient, inflexible

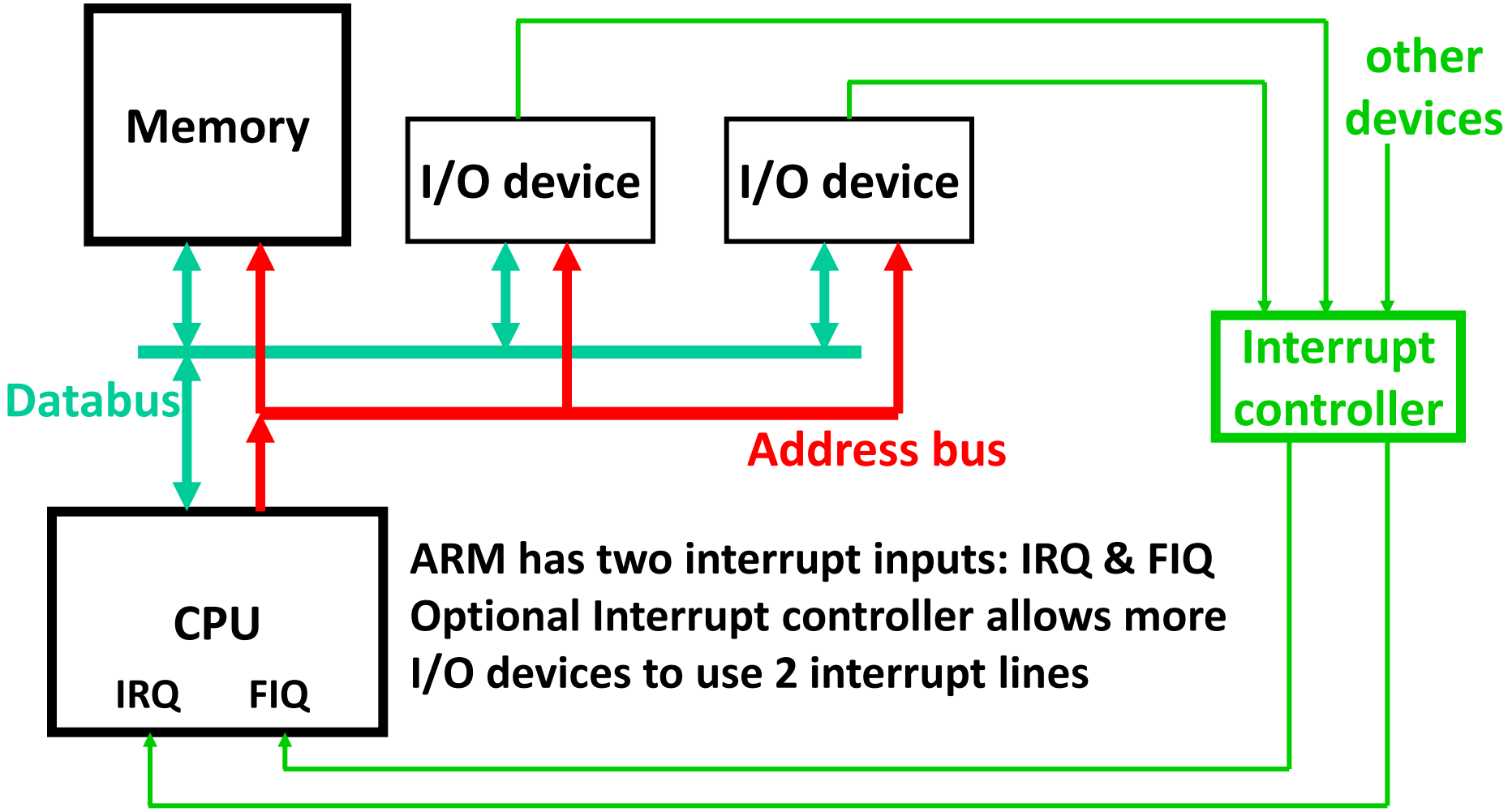
```
IO_SERVICE_CODE
        ADRL R10, SER_PORT          ; R10--> SERIAL_PORT
        ADR R11, BUFFER             ; R11-> array of bytes read
POLL_LOOP                           ; example of polling I/O
        LDR R0, [R10,#PORT_STATUS];input status
        CMP R0, #0
        BPL POLL_LOOP               ; wait till port is ready b31=1
        LDRB R0, [R10,#PORT_DATA]   ; input data
        STRB R0, [R11], #1          ; store data in memory
        B POLL_LOOP                 ; repeat
```

Method 2: Interrupt

- CPU executes user code independently of I/O
- I/O device raises interrupt line when it requires service
- CPU receives interrupt, suspends current execution, handles device in an interrupt service routine, then returns to previous execution as though nothing had happened
- Interrupt is invisible to executing code except is small loss of speed

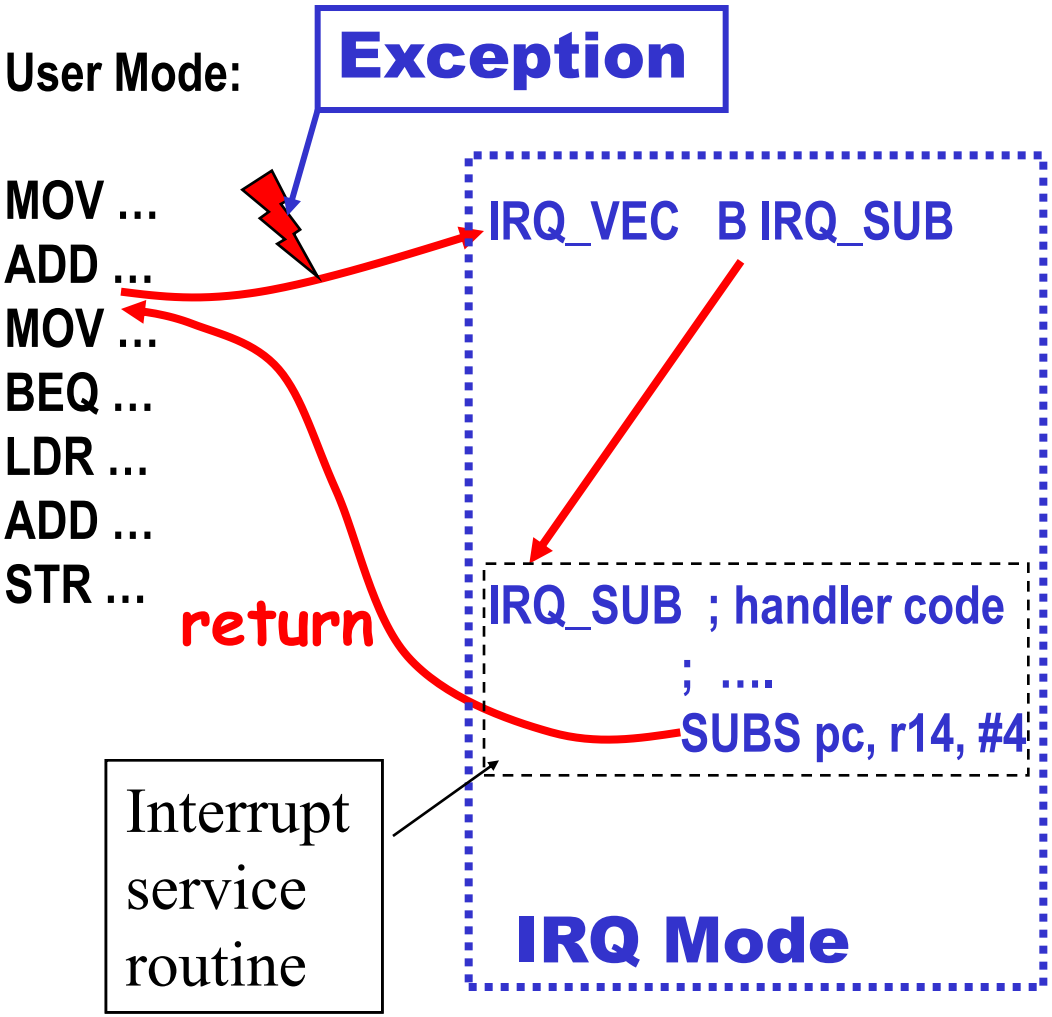
Interrupt Hardware for ARM

- I/O devices communicate via memory-mapped registers
- DMA & Interrupts are optional



Anatomy of an interrupt exception

- The **interrupt** causes user code to suspend and a branch to a fixed hardware determined memory location - so-called interrupt "vector" (labelled IRQ_VEC here)
  - CPU enters **IRQ mode**
- The vector contains a single branch to a software-determined **interrupt service routine** IRQ_SUB
- When the interrupt has been dealt with the ISR makes an interrupt return
- User code continues execution as though interrupt had never happened - no register is disturbed



ARM Operating Modes

- The ARM processor can work in one of many **operating modes**. So far we have considered **user mode**, which is the "**normal**" mode of operation.
- The processor can also enter "**privileged**" operating modes which are used to handle **exceptions** and **SWIs**
- The Current Processor Status Register **CPSR** has 5 bits [bit4:0] to indicate which mode the processor is in:-

Mode bits

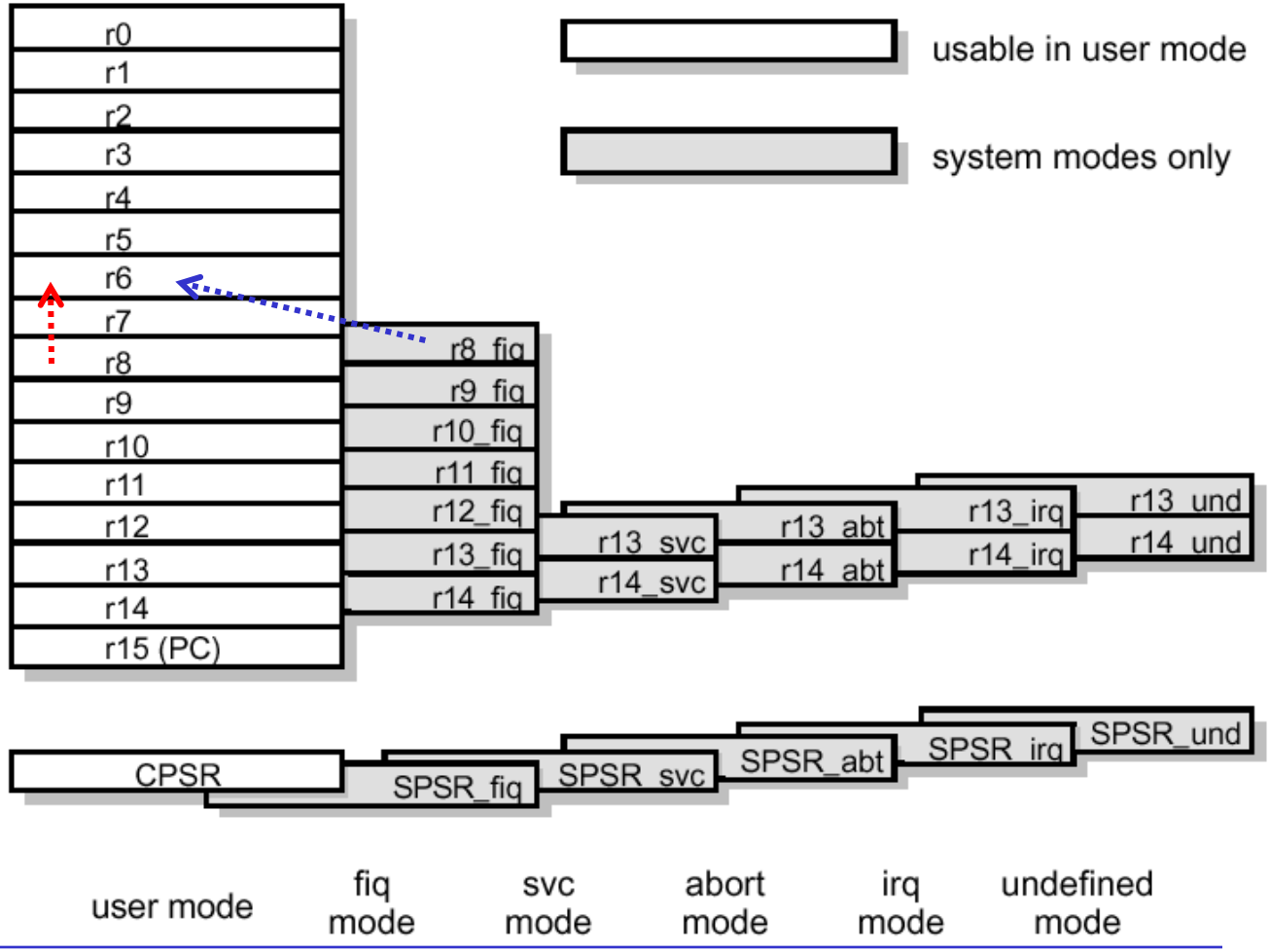
CPSR[4:0]	Mode	Use	Registers
10000	User	Normal user code	user
10001	FIQ	Processing fast interrupts	_fiq
10010	IRQ	Processing standard interrupts	_irq
10011	SVC	Processing software interrupts (SWIs)	_svc
10111	Abort	Processing memory faults	_abt
11011	Undef	Handling undefined instruction traps	_und
11111	System	Running privileged operating system tasks	user

Shadow Registers – Clever Feature of ARM ISA enables Exceptions

- As the processor enters an exception mode, some new registers are automatically switched in to avoid overwriting user regs:-

(FIQ Mode): MOV R6, R8

(User Mode):MOV R6, R8



Shadow Registers – Key points

- For example, an external event (such as movement of the mouse) occurs that generates a Fast Interrupt (on the FIQ pin), the processor enters FIQ operating mode.
- It sees the same r0 - r7 as before, and a new set of shadow r8 - r14
  - By swapping to the new registers, it is easier for the programmer to preserve the state of the processor.
    - For example, during FIQ mode, the FIQ versions of r8 - r14 can be used freely. On returning back to user mode, the original values of r8 - r14 will never have changed.
- The shadow r8-r14 preserve their values across FIQ interrupts and can be used to store **persistent FIQ state.**

ARM Exception Handler Routines

- r13, r14 are shadowed automatically
  - In handler the alternative shadow registers are used. User r13,r14 not changed.
  - **R13** - stack pointer
  - **R14** link register holds return address
- **User R1** is saved on ISR mode stack indexed by shadow r13 SP.
- R1 used for temp data in ISR -holds value of TIME
- Code increments TIME
- R1 is restored at the end of the ISR
- CPSR is saved/restored automatically

Exception_handler (not FIQ)

```
STMED r13!, {r1}
LDR r1, TIME
ADD r1, r1, #1
STR r1, TIME
LDMED r13!, {r1}
SUBS pc, r14, #4
```

ARM Exception Handler Routines

FIQ_handler_opt (FIQ)

```
1  ADD r8, r8, #1
4  STR r8, TIME
4  SUBS pc, r14, #4
```

9 cycles

Exception_handler (not FIQ)

```
4  STMED r13!, {r1}
4  LDR r1, TIME
1  ADD r1, r1, #1
4  STR r1, TIME
4  LDMED r13!, {r1}
4  SUBS pc, r14, #4
```

21 cycles

FIQ_handler (FIQ)

```
4  LDR r8, TIME ;
1  ADD r8, r8, #1
4  STR r8, TIME
4  SUBS pc, r14, #4
```

13 cycles

How are exceptions generated?

- **Classify exceptions according to cause:**
  - **1. As a *direct result of executing an instruction*, such as:**
    - **Software Interrupt Instruction (SWI)**
    - **Undefined or illegal instruction**
    - **Memory error during fetching an instruction**
  - **2. As a *side-effect of an instruction*, such as:**
    - **Memory fault during data read/write from memory**
    - **Arithmetic error (e.g. divide by zero if CPU has divide instruction)**
  - **3. As a *result of external hardware signals*, such as:**
    - **Reset (e.g. from reset switch)**
    - **Fast Interrupt (FIQ)**
    - **Normal Interrupt (IRQ)**

IRQ, FIQ are typically connected to hardware devices requiring interrupt service

What does an exception do?

- User code is executing, and an exception happens
- The handler runs like a subroutine, and at the end returns to the next user instruction (just like a subroutine) but also restores **all processor registers (including CPSR).** This makes the handler completely transparent to the User-mode code.
- The job of the handler is to deal with the condition that caused the exception:
  - Recover from memory fault
  - Perform SWI operation
  - Service the hardware device that caused the interrupt

What happens when an exception occurs?

- ARM completes current instruction as best it can.
- It departs from current instruction sequence to handle the exception by performing the following steps:-
  - 1. It changes the operating mode (see slide 3.11) corresponding to the particular exception.
  - 2. It saves the current PC in the R14 corresponding to the new mode. For example, if FIQ occurs, the PC value is stored in R14(FIQ). This will be the return address (but see slide 3.25)
  - 3. It saves the old value of CPSR in a special register of the new mode.
  - 4. It disables exceptions of lower priority
  - 5. It forces the PC to a new value corresponding to the exception. This is effectively a forced jump to the Exception Handler or Interrupt Service Routine.

CPSR during interrupts

- The CPSR must be preserved by interrupts
- ARM architecture contains a mechanism to do this, but it is not a shadow register.
  - On interrupt or exception the old CPSR (current PSR) **contents are saved** in an SPSR (saved PSR) register specific to the mode of the interrupt or exception, however the same CPSR continues to be used – its mode bits (see slide 3.11 are changed to the exception mode.
  - On return the saved value is written back to CPSR by hardware.
  - This is subtly different from the shadowing mechanism, since **the same register is used for CPSR in all modes**

Where is the exception handler routine?

- Exceptions can be viewed as "forced" subroutine calls.
  - When and if an exception occurs is not predictable (unless it is an SWI exception).
  - The address to which the processor is forced to branch to is called the **exception (or interrupt) vector**.
  - It is fixed by hardware.

Exception vector addresses

- Each vector (except FIQ) is 4 bytes long (i.e. one instruction) at a fixed position in low memory.
- You put a branch instruction at this address, e.g.:

```
B       svc_exception_handler
```

- See next slide

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

- **FIQ is special in two ways:-**
  - **1. You can put the actual FIQ handler (also called Fast Interrupt Service Routine) at 0x0000001C onwards, because FIQ vector occupies the highest address**
  - **2. FIQ has many more shadow registers. So you don't have to save as many registers on the stack as other exceptions - faster.**

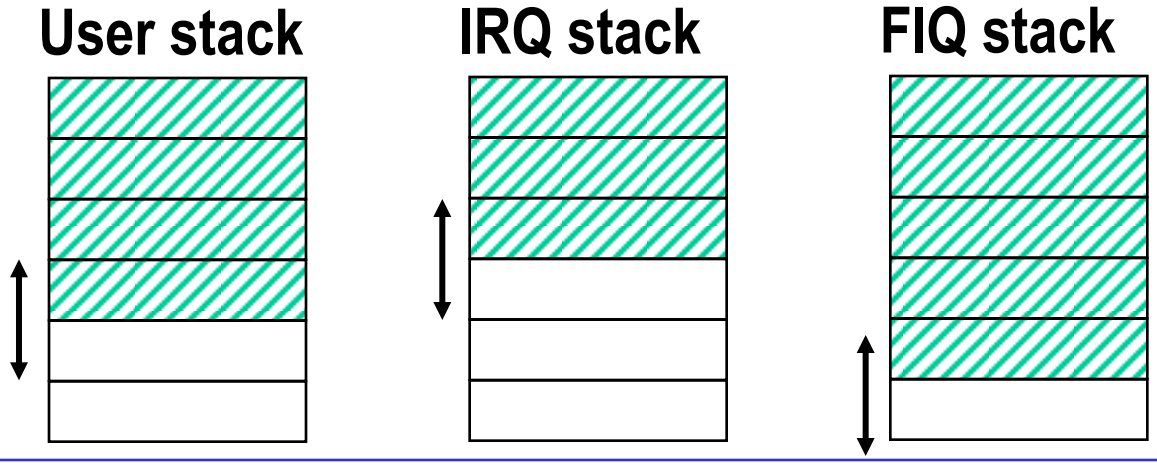
```
RST_VEC     B RESET_SUB
UND_VEC     B UND_SUB
SWI_VEC     B SWI_SUB
PRE_VEC     B PREFETCH_SUB
DATA_VEC    B DATA_SUB
IRQ_VEC     B IRQ_SUB           ; branch adds 4 cycles to handler time
FIQ_VEC     ADD R8, R8, #1      ; don't need a B here
            STR R8, TIME
            SUBS pc, r14, #4    ; return from interrupt

TIME        DCD 0               ; memory location for TIME variable
```

NB - why B not BL?

Multiple Stacks

- Exception modes replace user R13 by a shadow register which stores the SP of the exception stack. Separate shadow register for each mode => different stack for each mode.
- User registers can then be saved/restored safely on the exception stack



Exception Return

- Once the exception has been handled (by the exception handler), the user task is resumed.
- The handler program (or Interrupt Service Routine) must restore the user state exactly as it was before the exception occurred:
  - 1. Any user registers saved on the exception handler's stack must be restored from it
  - 2. The CPSR must be restored from the appropriate SPSR – *(done by processor automatically).*
  - 3. PC must be changed back to the instruction address in the user instruction stream
- Steps 1 and 3 are done explicitly by exception handler code
- Restoring registers from the stack (and saving them initially) would be the same as in the case of subroutines
- Restoring PC value is more complicated. The exact way to do it depends on which exception you are returning from.

Exception Return (con't)

- Remember that the return address was saved in r14 before entering the exception handler.
- To return from a SWI or undefined instruction trap, use:

  MOVS pc, r14

- To return from an IRQ, FIQ or prefetch abort, use:

  SUBS pc, r14, #4

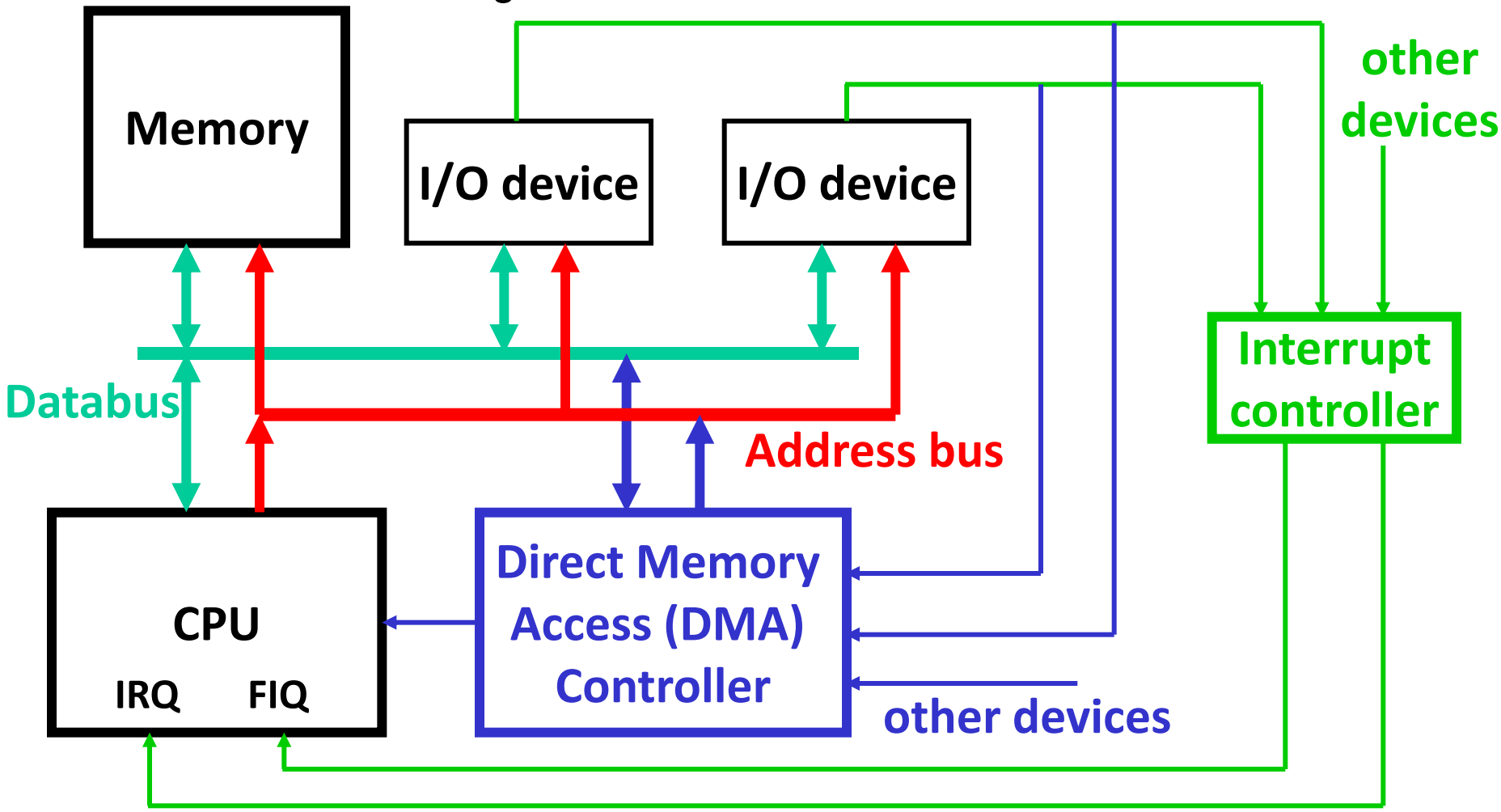
- To return from a data abort to retry the data access instruction that failed, use:

  SUBS pc, r14, #8

- If the destination register is the PC, the 'S' modifier does NOT mean "set the flags", but "restore the CPSR from the SPSR"
- The differences between these three methods of return is due to the **pipeline architecture** of the ARM processor.

Direct Memory Access vs Interrupt & Polling

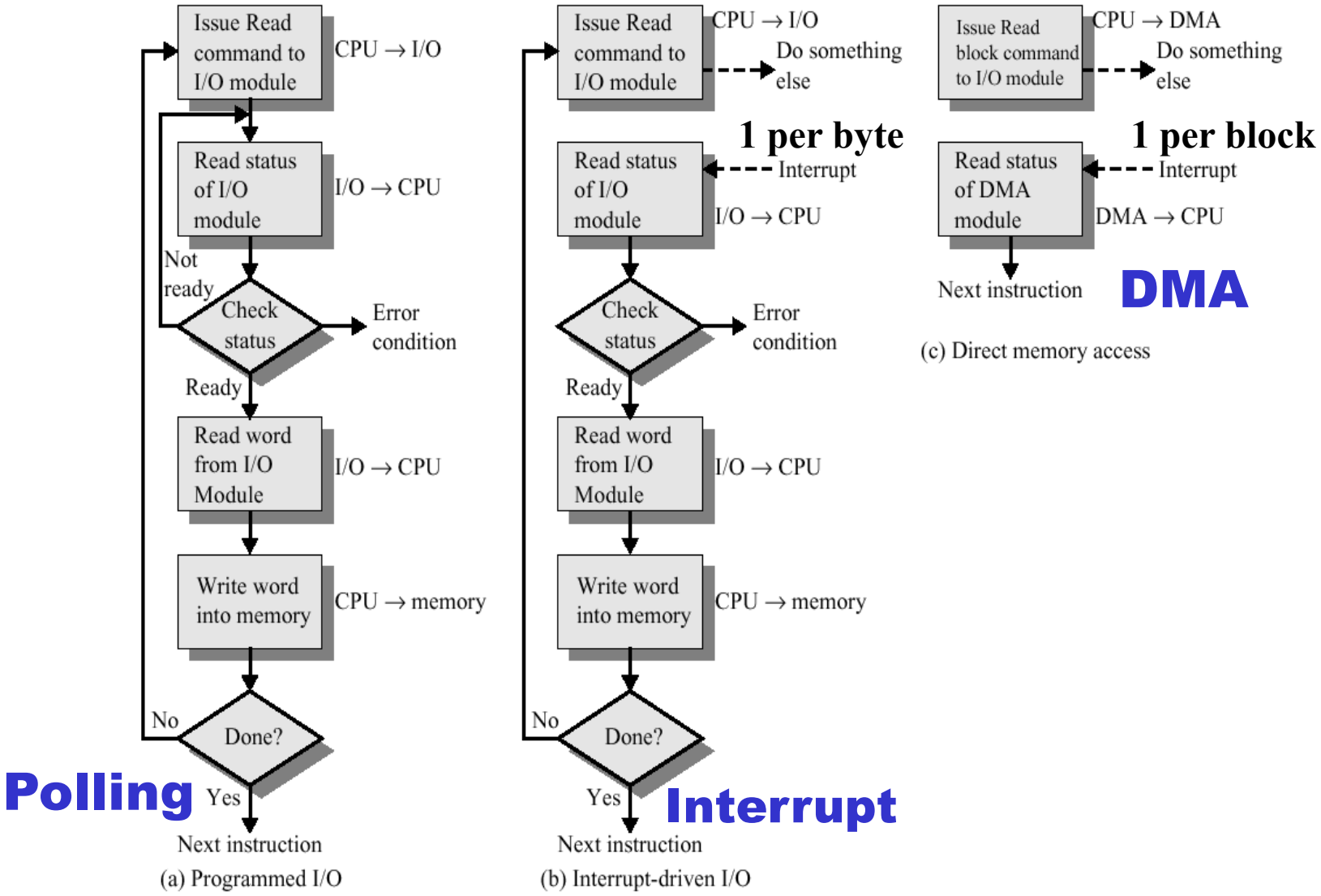
- DMA controller shares memory bus cycles with the processor - a technique known as **cycle stealing**. The CPU notices only a slightly slower memory system.
- DMA controller manages transfers



Direct Memory Access (con't)

- DMA is initiated by a processor. The following information must be sent by the processor to the DMA module (AKA DMA controller):
  - I/O device Read or Write is required?
  - Memory address of the I/O device involved in the transfer
  - The starting location in memory to read from or write to
  - The number of word to read or write
- Once DMA is initiated, the processor can continue with other work.
- Processor can work concurrently with transfer between I/O device and memory.
- Data transfer without interrupt overhead

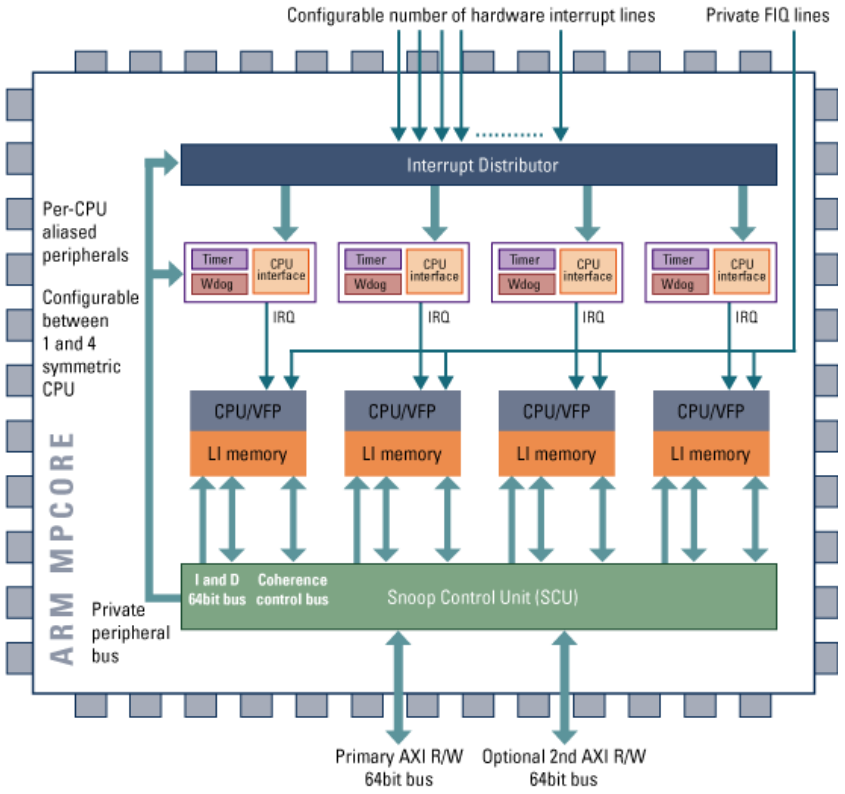
Techniques for Inputting a Block of Data: DMA



Lecture 13 ARM Processor Organization

"It's a poor bureaucrat who can't stall a good idea until even its sponsor is relieved to see it dead and officially buried" Robert Townsend

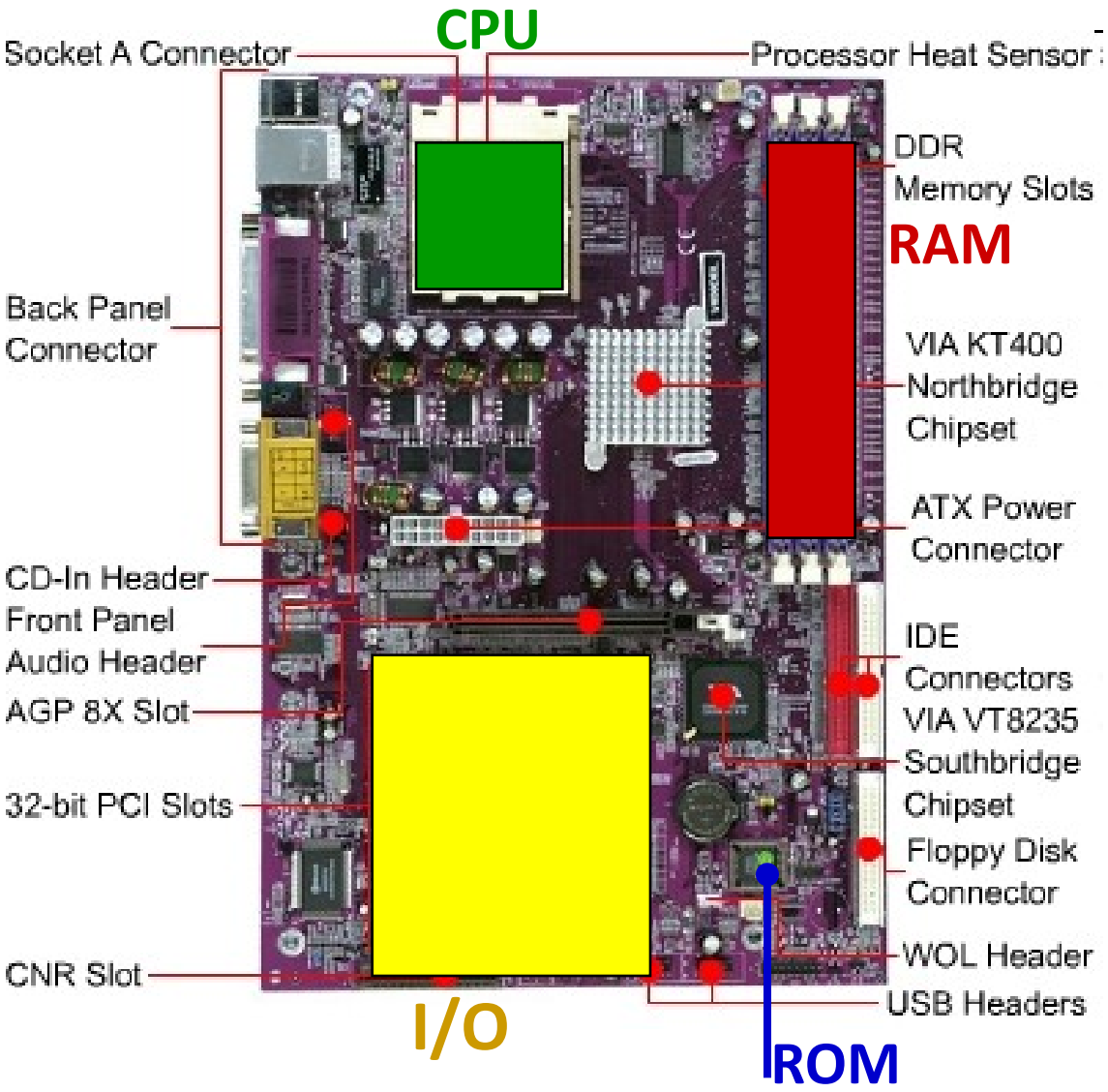
- First ARM processor developed on 3 micron technology in '83-'85
- This course is mainly based on the ARM6/7 architecture developed between '90-'95 and now the market leader fior low/medium performance embedded applications.
  - ARM has continued development. Recent faster compute engines:
    - ARM9: 220MIPS at 200Mhz clock, 0.2mW/Mhz
    - ARM10, 400+ MIPS
    - ARM11MP (4 processors, 2600MIPS)!
- Intel has developed an ARM-based Xscale architecture for PDAs etc.
- Later designs use 0.13µ technology: 25X smaller than the first ARM!



ARM11-MP multiprocessing core

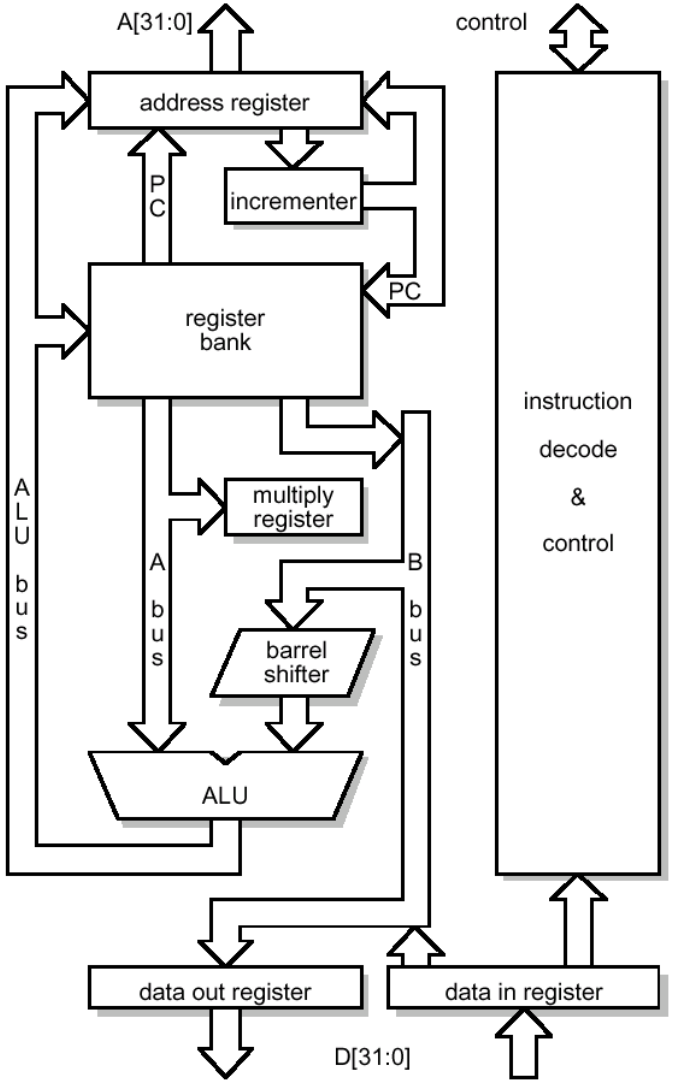
What is a computer?

- **Microcontroller**
  - ROM,RAM,CPU, peripherals all on one chip
  - Address, data,control bus inside chip
- **PC**
  - CPU on one chip (+ cache, see later)
  - ROM (BIOS chip)
  - RAM DDR2 DRAM
  - Address,data,control bus on motherboard
  - Peripherals, on motherboard or in PCI slots



Internal Organization of ARM CPU

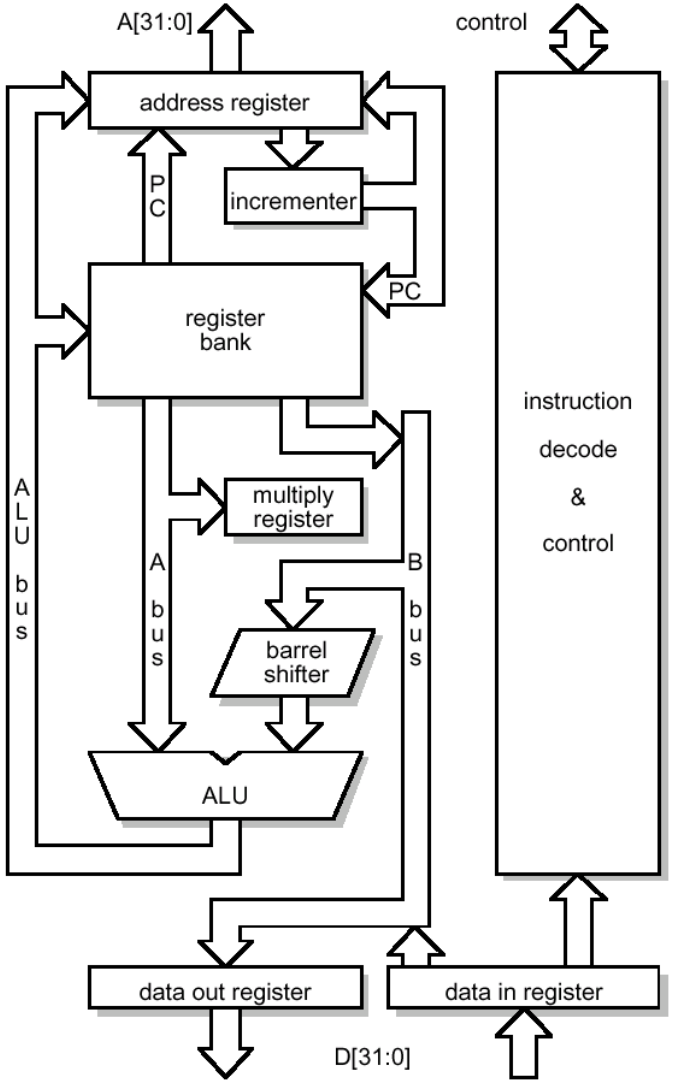
- Two main blocks: **datapath** and **decoder**
- Register bank (r0 to r15)
  - Two read ports to A-bus/B-bus
  - One write port from ALU-bus
  - Additional read/write ports for program counter r15
- Barrel shifter - shift/rotate 2nd operand by any number of bits
- ALU performs arithmetic/logic functions
- Dedicated PC incrementer
- Address register – either from PC or from ALU



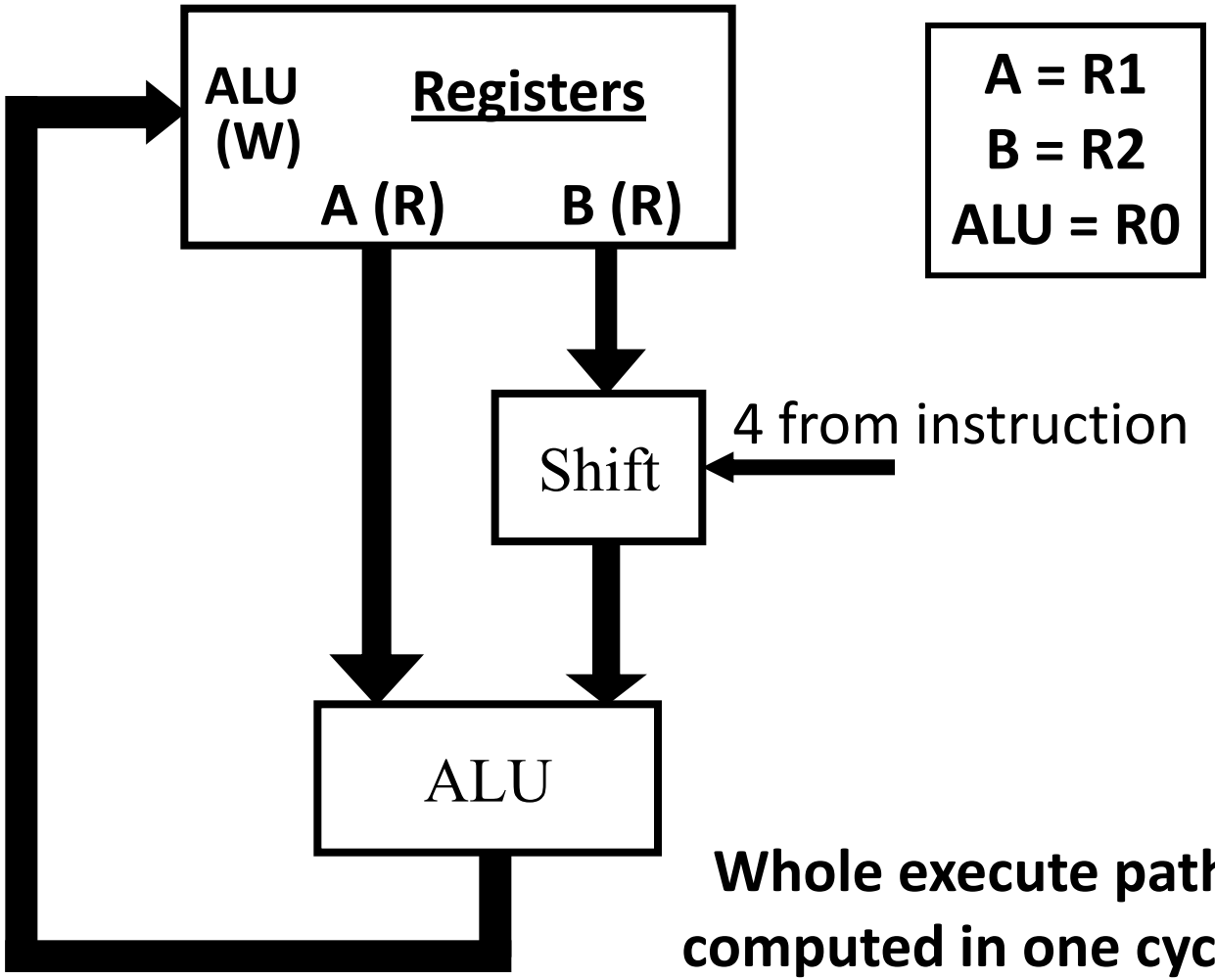
Internal Organization of ARM (con't)

- Data register holds read/write data from/to memory
- Instruction decoder decodes machine code instructions to produce control signals to datapath
- Data processing instructions take a single cycle: data values are read on the A-bus & B-bus, the results from ALU is written back into register bank

Can you see why shifts by the value of a register:
r0 := r1 + (r2 lsl r3)
take an extra cycle?



ADD R0, R1, R2, lsl #4



Register bank can read A,B registers, write ALU register, simultaneously Register nos A,B,ALU come from instruction

Whole execute path computed in one cycle

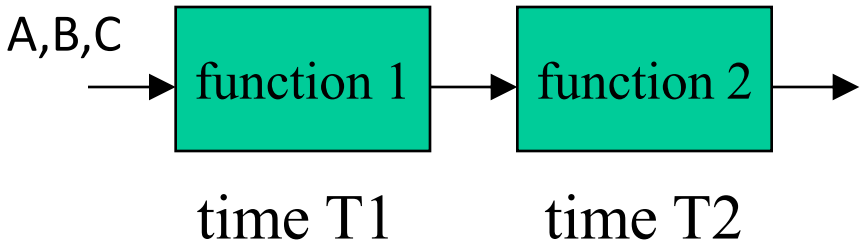
Speeding up CPU execution

- All computers typically perform a **sequence** of operations in each instruction:
  - Fetch instruction word: **FETCH**
  - Decode it (work out what to do): **DECODE**
  - Fetch register values
  - Perform ALU operation
  - Write back result to registers

  } **EXECUTE (all one cycle on ARM)**
- Each of these operations is done by different hardware, and doing them in strict sequence means that each hardware block would spend most of its time waiting for others
- This motivates the concept of **pipelining** – used by all modern CPUs

Pipelining

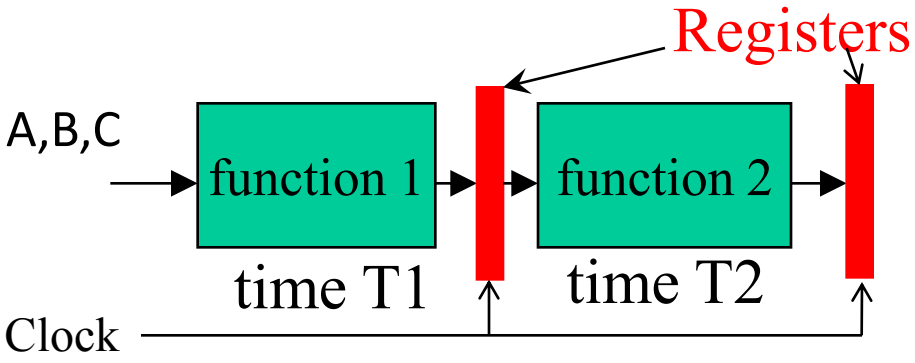
- The maximum processing rate is determined by the propagation delay of the computational logic – in abstract:



Timestep T1+T2

In	Out
A	f2(f1(A))
B	f2(f1(B))
C	f2(f1(C))

- Above we can process one input every T1+T2



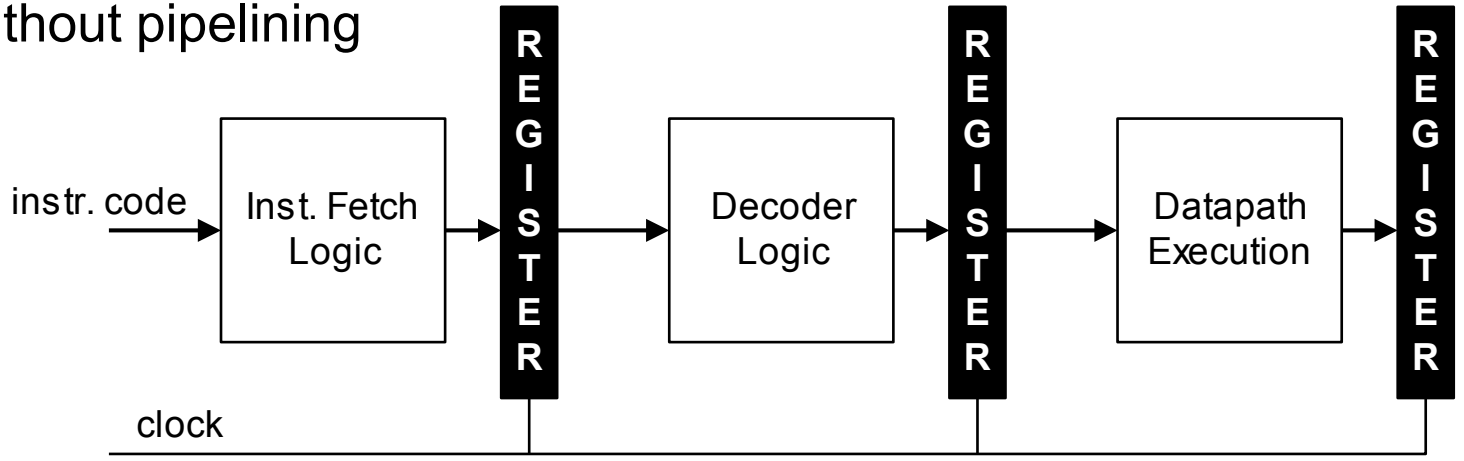
Timestep max(T1,T2)

In	Reg1	Reg2
A		
B	f1(A)	
C	f1(B)	f2(f1(A))
	f1(C)	f2(f1(B))
		f1(f2(C))

- Above we can process a new input every max{T1,T2}, but each input takes 2*max(T1,T2) > T1+T2 to be completely processed

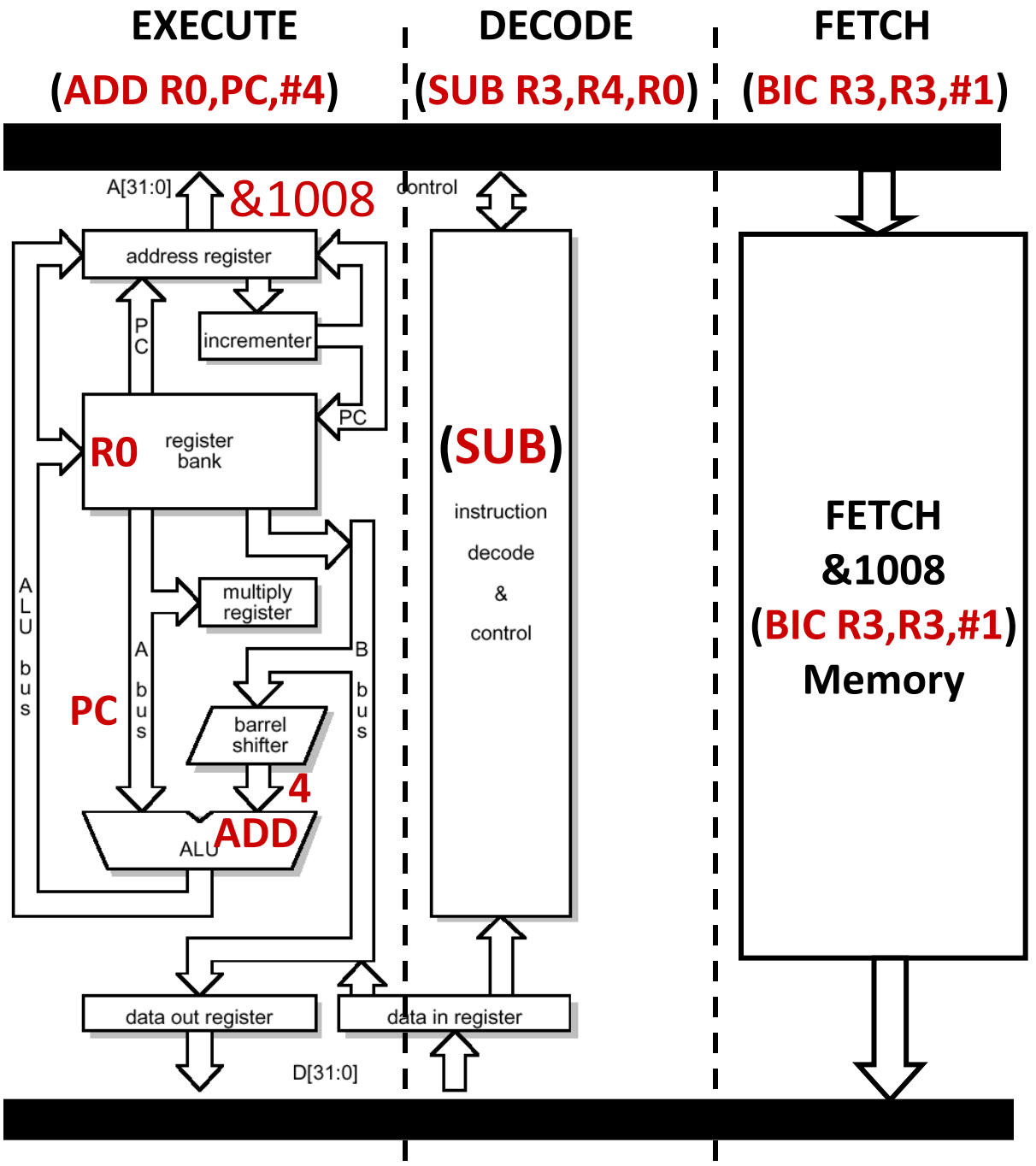
ARM Pipelining

- ARM core uses a 3-stage instruction pipeline
  - **Fetch**: **fetch instruction code from memory into the instruction pipeline**
  - **Decode: instruction decoded to obtain control signals for the datapath ready for the next stage**
  - **Execute: instruction "owns" the datapath - register read; shifting; ALU results generated and write-back, all in one cycle.**
- Results for each stage stored in registers
- The consequence is that the clock period is much shorter than without pipelining

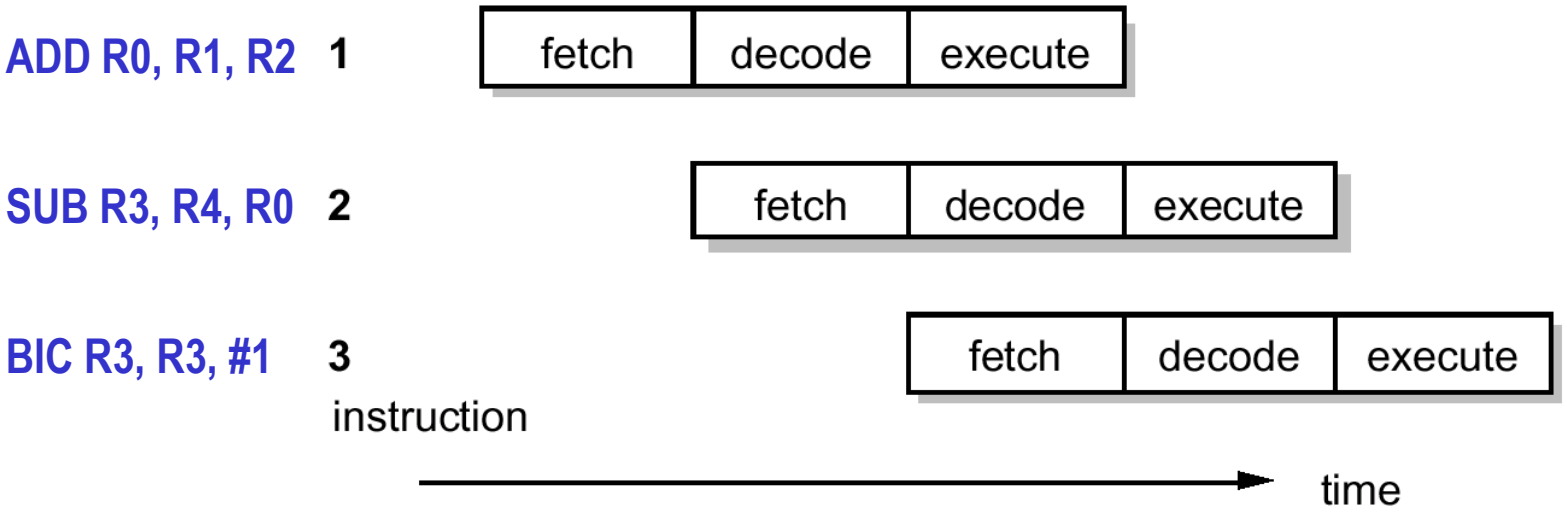


ARM CPU when ADD instruction is executing

&1000 ADD R0,PC, #4
&1004 SUB R3, R4, R0
&1008 BIC R3, R3, #1



ARM Pipelining (con't)



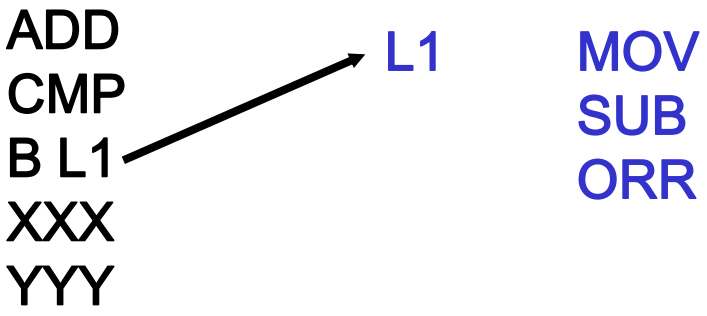
- At any time, 3 different instructions may occupy each of the the 3-stages of pipeline
- It may take three cycles to complete a single-cycle instruction. This is said to have a three cycle **Latency**
- Once a pipeline fills, the processor completes a single-cycle instruction every clock cycle. Therefore the **throughput (T)** is one instruction per cycle.

Branches & pipeline stalls

- Pipelining is a key design technique.
  - For example the Intel Pentium 4 pipeline was 20 stages long!
  - This is excessive, and this design was abandoned in the later Pentium Mobile core
  - AMD new 64 bit Hammer Athlon architecture uses 10/12 stages
- Long pipelines mean that many future instructions must start processing before the current instruction has finished
- Problems occur when branches happen changing the future course of instruction execution
  - Unconditional branch may be OK – extra fetch hardware can check for and follow branches - though ARM does not have this
  - Conditional branches are a real problem – cannot tell which way to go!
  - Cost of wrong branch prediction is a **pipeline stall** waiting for the correct instructions to fill up the pipeline

Pipeline stalls due to branches on ARM

- ARM pipeline will stall **whenever a branch is taken**
- Stall is detected at end of EXECUTE cycle of Branch instruction
- Stall costs 3 cycles in addition to normal one cycle execution.
  4. *One cycle (memory wait) to load non-contiguous FETCH address*
     - This is extra cost due to random access to memory being one cycle slower than typical sequential access
  5. One cycle to fill FETCH pipeline stage
  6. One cycle to refill DECODE pipeline stage
  7. After which the EXECUTE cycle of the correct instruction can be performed.



cycle	FETCH	DECODE	EXECUTE
1	B L1	CMP	ADD
2	XXX	B L1	CMP
3	YYY	XXX	B L1
4	*memory wait*	stalled	stalled
5	**MOV (at L1)**	stalled	stalled
6	**SUB**	**MOV**	stalled
7	**ORR**	**SUB**	**MOV**

ARM Terminology

- Instruction is **fetched** if it enters FETCH stage of pipeline (whether or not it ever reaches EXECUTE stage)
- Instruction is **executed** only if it reaches EXECUTE stage of pipeline
- When instruction is executed but execution condition is false - so nothing happens - we say it is **condition false executed**
- When instruction is executed and execution condition is true we say it is **condition true executed**.
  - B XXX is always **condition true executed** if it is executed
  - BEQ XXX is **condition true executed** if Z=1
  - ADDNE R0,R1,R2 is condition true executed if Z=0
- Note that a condition false executed instruction is inspected and condition checked in EXECUTE stage - unlike an instruction which never reaches EXECUTE stage of pipeline, but may be FETCHED or DECODED

Example execution

```
    ; throughout Z=1, EQ true, NE false
    ADD R0,R1,R2          ;Condition true executed
    ORRNE R5,R5,R5        ;Condition false executed
    RSBEQ R3,R3,R3        ;Condition true executed
    EOREQ R5,R5,R6        ;Condition true executed
    BNE A                 ;Condition false executed
    BEQ A                 ;Condition true executed
    SUB R2,R3,R4          ;Fetched, decoded, not executed
    ADC R1,R0,R0          ;Fetched, not decoded or executed

A   AND R2,R3,R4          ;Condition true executed
```

Speed loss through ARM stalls

- On ARM7 processor 3 cycles are lost whenever branch is executed.
- Assume one instruction = one cycle except for pipeline stalls
  - Nearly correct
- Suppose 20% of instructions are **condition true executed** branches (ie they actually happen):
  - B
  - BEQ (if Z=0)
  - BL
  - MOV pc,r14
- Calculate loss due to stall:
  - Every 5 instructions = 5 cycles without stall there will be one stall = 3 extra cycles lost.
  - Throughput = 5/8 = 0.625 instructions/cycle.
    - If clock = 60MHz, throughput = 37.5 MIPS (million instructions per second)

Stalls & Branch Prediction

- Stalls happen when wrong instruction is prefetched.
- This can be avoided by **branch prediction** - work out whether branch will happen and if it will fetch the branch target address
  - Stall only if branch instruction **and branch is not correctly predicted**
- Three types of branches
  - ARM predicts correctly only condition false executed branches!

Type of branch	**ARM examples**	**ARM prediction**
Unconditional Branch	B XXX BL XXX	100% Incorrect
Conditional Branch	BEQ XXX BLEQ XXX	Correct if branch not taken Incorrect if branch taken
Computed Branch	MOV pc, R14	100% Incorrect

Throughput Calculation

- Assume
  - 1 instruction/cycle when pipeline full
  - Branches happen with probability *B* per instruction
  - Branches lead to stall only if **not correctly predicted**
    - *e.g. in ARM branch condition false executed has no stall because this is predicted by ARM hardware, but branch condition true executed leads to stall.*
  - *Incorrect* branch Prediction has probability P
  - Pipeline penalty from incorrect branch prediction is a **pipeline Stall** of *S* cycles.
  - Typically assume *S* ~ length of pipeline (not exact but good guess)
- Average number of cycles **added** per instruction is: *P*B*S*
- Throughput (per cycle) decreases to:

$$T = \frac{1}{1 + PBS} \text{ instructions/cycle}$$

- Multiply T by clock frequency for throughput in instructions/second
  - MIPS - millions of instructions/second

Summary

- ARM is *pipelined*
  - throughput of up to 1 cycle per instruction
  - latency of 3 cycles per instruction
- Most instructions take one cycle
- Exceptions relate to hardware

Instruction	Cycles	
Add R0, R1, R2, lsl R3	2	1 cycle added to read R3 since only three register file ports
B, BL, **MOV pc, R14**	4	Pipeline stall & memory wait adds 3 cycles
LDR/STR	4	Memory read/write (& two memory waits) add 3 cycles
LDM/STM	3+n	n memory read/write (& two memory waits) adds 2+n cycles

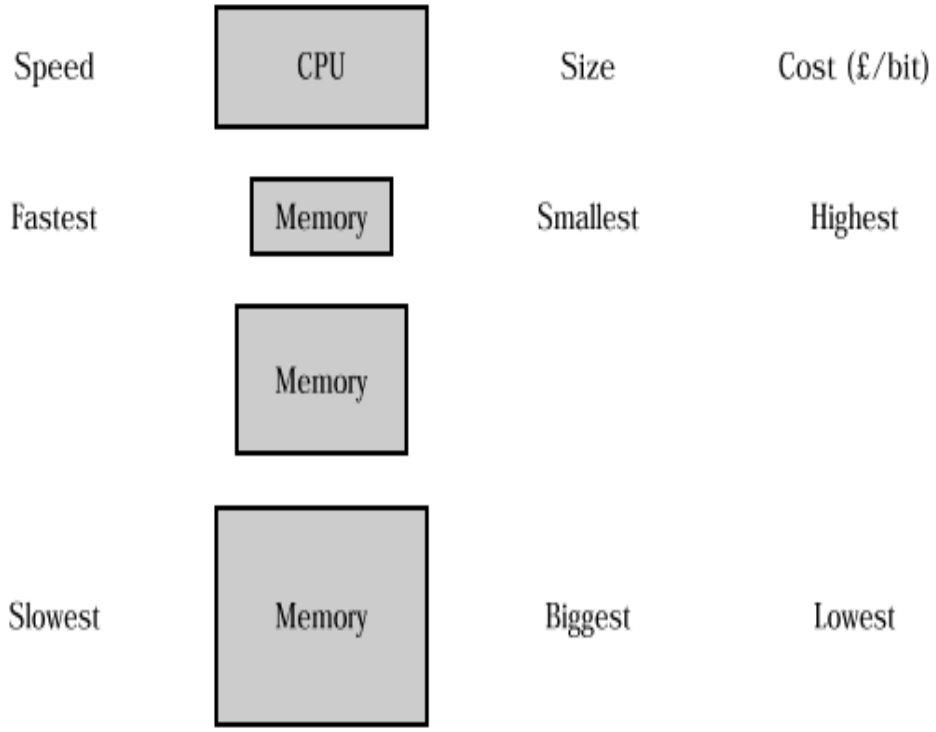
Lecture 14 - Cache Memory

"I suffer from short term memory loss. It runs in my family… At least I think it does… Where are they?" Finding Nemo

- **SRAM**:
  - value is stored on a pair of inverting gates
  - very fast but takes up more space (4 to 6 transistors per bit) than DRAM
- **DRAM**:
  - value is stored as charge on a capacitor (must be refreshed)
  - very dense (=> high capacity) but slower than SRAM (factor of 5 to 10)
- **Users want large and fast memories**!
  - SRAM access times are 2 - 20 ns at cost of $50 to $100 per Mbyte
  - DRAM access times are 10-30 ns at cost of $0.05 to $0.2 per Mbyte
  - Disk access times are 5 - 10 million ns at cost < $0.001 to $0.003 per Mbyte

Exploiting Memory Hierarchy

- Question:
  - how to organize memory to improve performance without the cost?
- Answer:
  - build a **memory hierarchy**
  - Store copies of frequently used memory locations in smaller & faster memories close to the CPU
  - Memory reference instructions use the closest memory containing the required item
  - Can speed up **average** memory access manyfold.

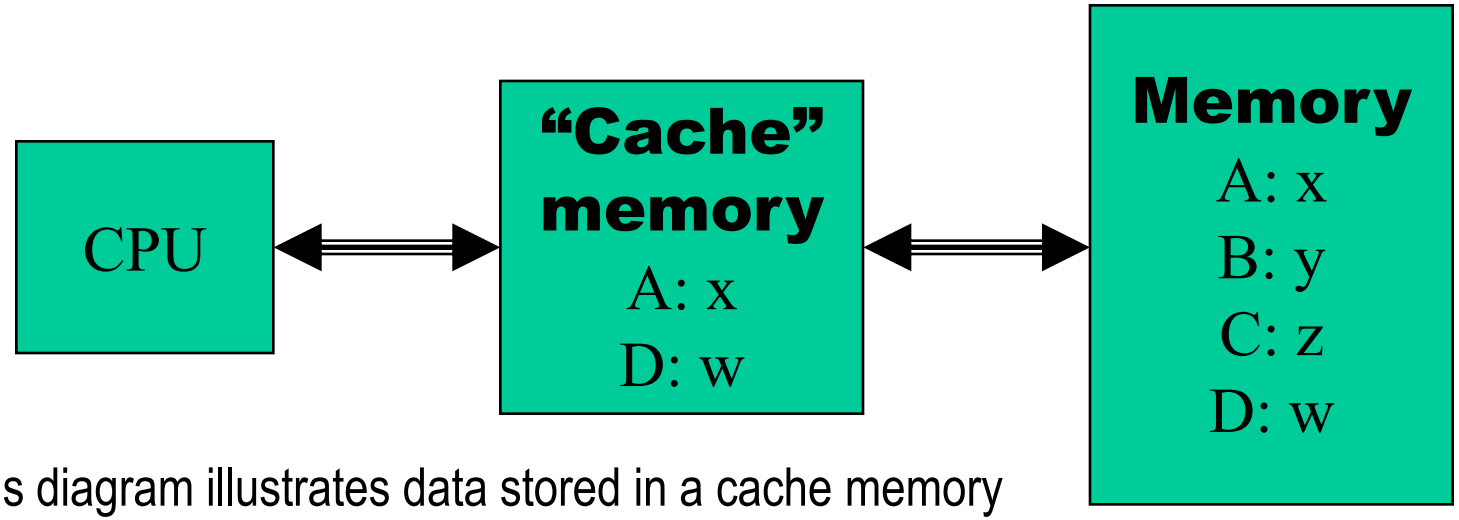


Exploiting Memory Hierarchy

memory type		size	access speed
Processor Registers	*CPU controlled transfer*	64 to 256 bytes	0.3 - 5 nsec
On-chip cache	***Automatic (hardware-controlled) transfer***	8 - 1024 Kbytes	3-5 nsec
Second-level cache		256 - 16384 Kbytes	5-10 nsec
Main memory (DRAM)		16M - 16G bytes	~ 20 nsec
Disk or other store	*CPU controlled transfer*	10's - 1000's Gbytes	10's msec

nearest to CPU

Automatic memory transfer between levels of hierarchy



- This diagram illustrates data stored in a cache memory and a larger memory further from the CPU.
  - Location A has value x stored in Cache etc.
- The cache stores some, but not all, of the items in the larger memory.
- Here addresses A & D, if requested by the CPU, will be HITS in the cache, and addresses B & D MISSes.
- In a memory hierarchy every level looks like this this, requests from the CPU move outwards till they HIT.

Cache Terminology

- Memory **hierarchy** can be multiple levels
- Data is copied between two adjacent levels at a time
- We will focus on two levels:
  - **Upper level** (closer to the processor, smaller but faster)
  - **Lower level** (further from the processor, larger but slower)
- Some terms used in describing memory hierarchy:
  - **block**: minimum unit of data to transfer between levels - also called a **cache line**
  - **hit**: data requested is in the upper level
  - **miss**: data requested is not in the upper level

The Basics of Caches

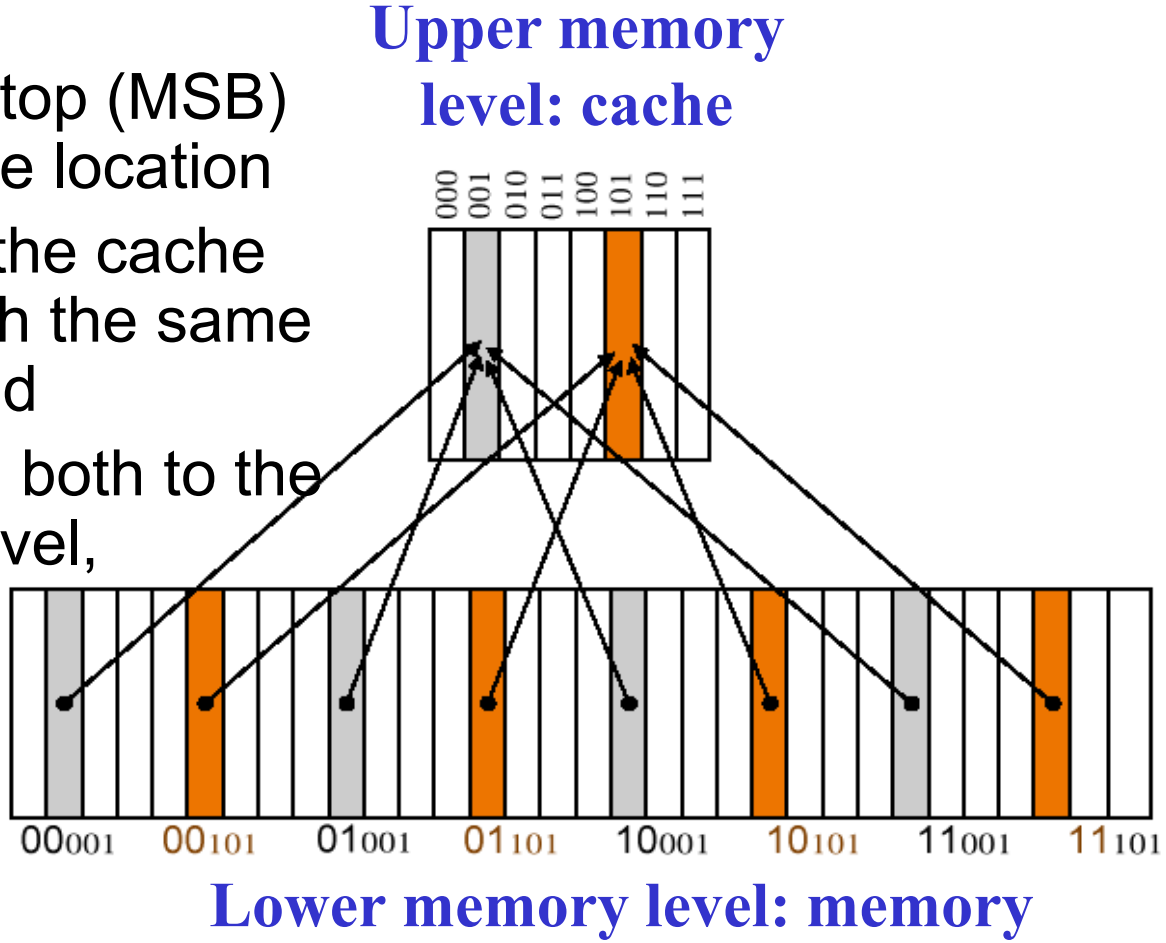
- Two issues:
  - How do we know if a data item is in the cache? (Is it a **HIT** or a **MISS**?)
  - If it is there (a HIT), **how do we find it**?
- Our first example:
  - block size is one byte of data
  - we will consider the "**direct mapped**" approach

For each item of data at the lower level,
there is exactly one location in the cache memory where it might be.

Lots of items at the lower level share a given location in the upper level, which can contain at most one at any given time.

Direct Mapped Cache

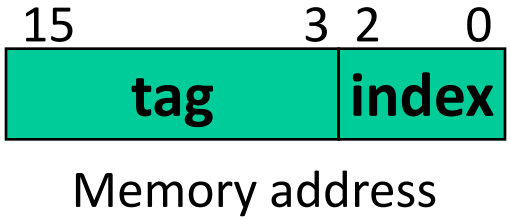
- Mapping: Bottom (LSB) bits of the address determine position in cache
- Items differing only by top (MSB) bits map to same cache location
- An item is replaced in the cache when different item with the same bottom bits is requested
- Assume data is written both to the cache and the lower level, preserving coherence
  - **Write-through** cache



Cache Operation

Cache Address	Cache Data		
Index	Data	Tag	V
000			0
001			0
010			0
011			0
100			0
101			0
110			0
111			0

- **Cache address** = index
- Each cache line (block) consists of Data, Tag & Valid fields
- **Index** part of **memory address** determines which cache line is used.
- **Tag** part of **memory address** is matched with stored tag in cache



Memory address

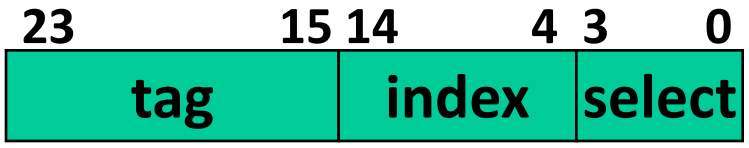
Cache data line

- Data field of cache contains cache line, can be more than one byte of memory
- **Select** bits from memory address are used to select byte within the cache line

2^4 = 16 bytes per line

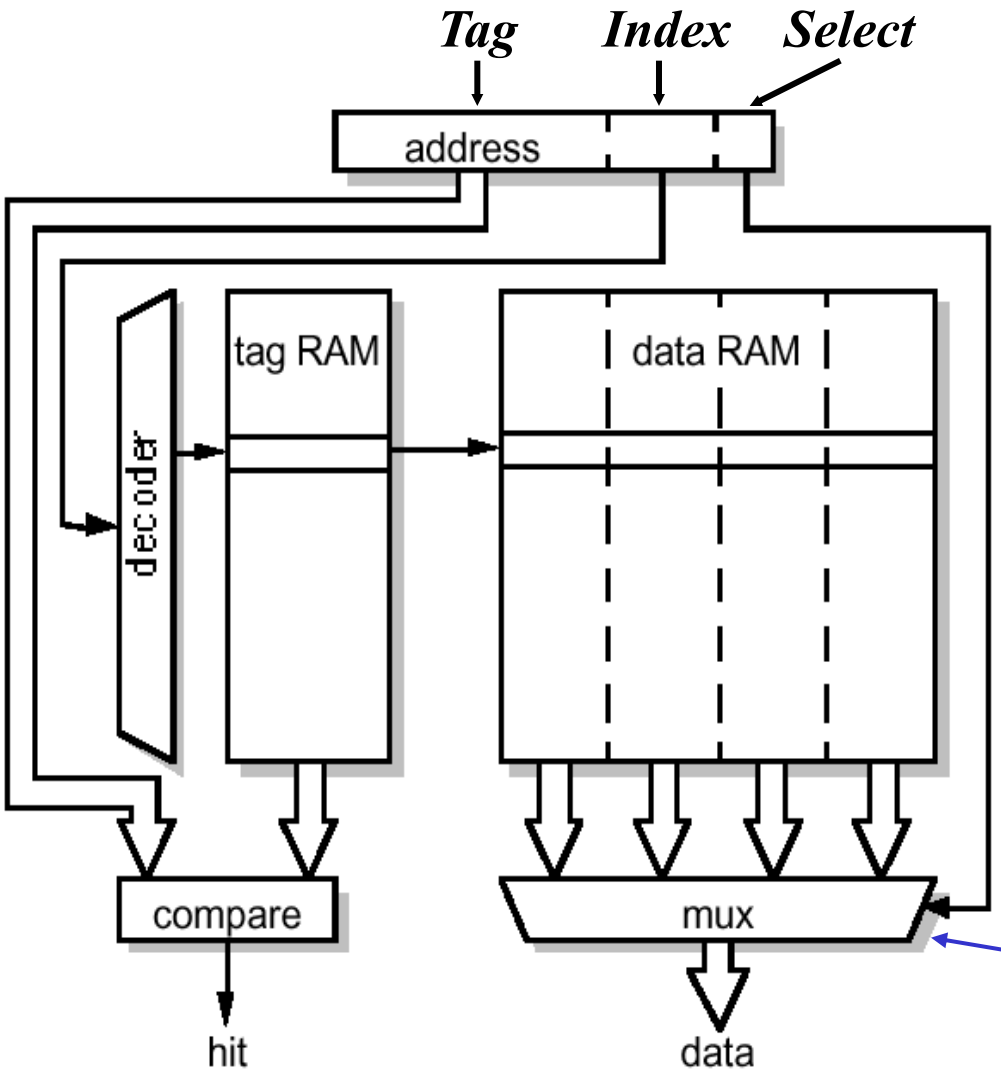
2^{11} = 2048 lines

2^{15} = 32768 bytes cache memory



Memory address

Organization of Direct-mapped cache



n bit *Index* selects 1 out of 2^n cache lines

m bit *Select* selects 1 out of 2^m data bytes within a line

Cache Contents - A walk-through
Assume cache is 8 blocks (lines) each of 1 byte

Index	Valid bit (V)	Tag	Data
000	0		
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

- *Initial state on power-ON*
- After handling read of address 00**000**
- After handling read of address 00**001**
- After handling write to address 01**010**
- After handling read of address 01**000**
- After handling read of address 01**010**

Cache Contents - A walk-through (2)

Index	Valid bit (V)	Tag	Data
000	1	00	MEM[00000]
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

- Initial state on power-ON
- *After handling read of address 00**000, miss, mem read***
- After handling read of address 00**001**
- After handling write to address 01**010**
- After handling read of address 01**000**
- After handling read of address 01**010**

Cache Contents - A walk-through (3)

Index	Valid bit (V)	Tag	Data
000	1	00	MEM[00000]
001	1	00	MEM[00001]
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

- Initial state on power-ON
- After handling read of address 00**000, miss, mem read**
- *After handling read of address 00**001, miss, mem read***
- After handling write to address 01**010**
- After handling read of address 01**000**
- After handling read of address 01**010**

Cache Contents - A walk-through (4)

Index	Valid bit (V)	Tag	Data
000	1	00	MEM[00000]
001	1	00	MEM[00001]
010	1	01	Written MEM[01010]
011	0		
100	0		
101	0		
110	0		
111	0		

- Initial state on power-ON
- After handling read of address 00**000, miss, mem read**
- After handling read of address 00**001, miss, mem read**
- *After handling write to address 01**010,*** **miss, mem write**
- After handling read of address 01**000**
- After handling read of address 01**010**

NB - would write to MEM even if it was a hit

Cache Contents - A walk-through (5)

Index	Valid bit (V)	Tag	Data
000	1	01	MEM[01000]
001	1	00	MEM[00001]
010	1	01	MEM[01010]
011	0		
100	0		
101	0		
110	0		
111	0		

- Initial state on power-ON
- After handling read of address 00**000, miss, mem read**
- After handling read of address 00**001, miss, mem read**
- After handling write to address 01**010, miss, mem write**
- *After handling read of address 01**000, miss, mem read***
- After handling read of address 01**010, hit**

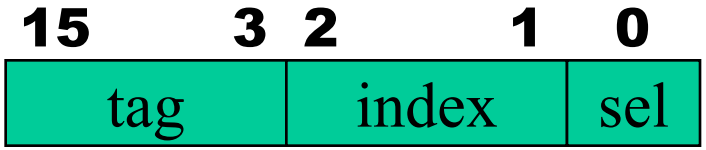
Cache Contents - A walk-through (6)

Index	Valid bit (V)	Tag	Data
000	1	01	MEM[01000]
001	1	00	MEM[00001]
010	1	01	MEM[01010]
011	0		
100	0		
101	0		
110	0		
111	0		

- Initial state on power-ON
- After handling read of address 00**000, miss, mem read**
- After handling read of address 00**001, miss, mem read**
- After handling write to address 01**010, miss, mem write**
- After handling read of address 01**000, miss, mem read**
- *After handling read of address 01**010, hit***

A Further Example

- We have considered a cache, consisting of 8 lines each of 1 byte. (Total 8 bytes).
- An alternative arrangement for 8 bytes **cache size** is 4 lines, each of 2 bytes.
- On each cache miss, we must then fill the entire line (on both writes and reads).
- The **sel** (select) field selects which byte in a line is addressed



- **Read hits**
  - This is what we want: cache passes data to processor
- **Read misses**
  - CPU stalls, fetch block from memory, deliver to cache, restart
- **Write hits:**
  - Can replace addressed data in cache and memory (write-through)
- **Write misses:**
  - Read the entire block into the cache line, then write the addressed data in cache and memory (write-through).
  - Why does line size 1 not require the read?

Cont'd…

- After handling read of address 00**00**0 (read block 00**00**) MISS

Index	Valid bit (V)	Tag	Data(0)	Data(1)
00	1	00	MEM[0]	MEM[1]
01	0			
10	0			
11	0			

- After handling read of address 00**00**1 HIT
  - No change from above

Each memory block is 2 bytes
The **index** bits are in bold text

- After handling write to address 01**01**0 (read block 01**01**, write block 01**01**) MISS

Index	Valid bit (V)	Tag	Data(0)	Data(1)
00	1	00	MEM[0]	MEM[1]
01	1	01	Write data	MEM[&0b]
10	0			
11	0			

- After handling read of address 01**01**1 HIT
  - No change from above

- After handling read of address 11**01**0 (read block 11**01**) MISS

Index	Valid bit (V)	Tag	Data(0)	Data(1)
00	1	00	MEM[0]	MEM[1]
01	1	11	MEM[&1a]	MEM[&1b]
10	0			
11	0			

Principle of Locality

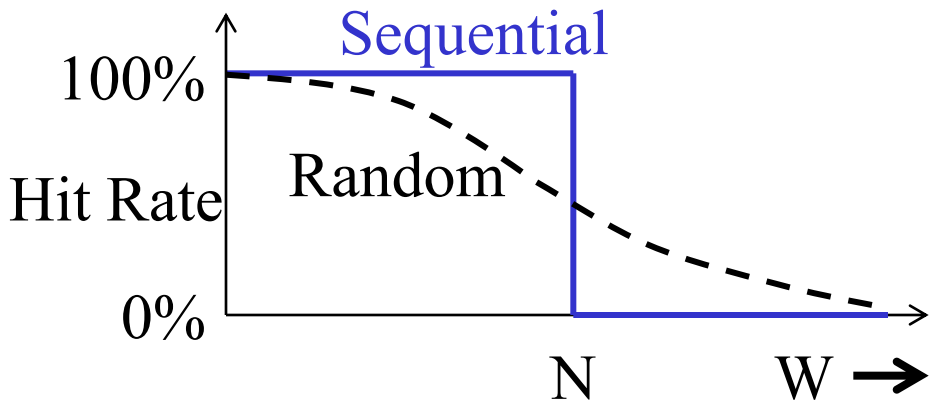
- The principle of locality makes having a memory hierarchy a good idea
- If an item is referenced,
  - **temporal locality**: it will tend to be referenced again soon
  - **spatial locality**: nearby items will tend to be referenced soon. (E.g. sequential access).
- **Why does code have locality?**
- All caches exploit temporal locality.
- A cache with a larger line size will achieve a higher hit rate for spatial locality
  - E.g. line size N items, sequential access => hit rate at least

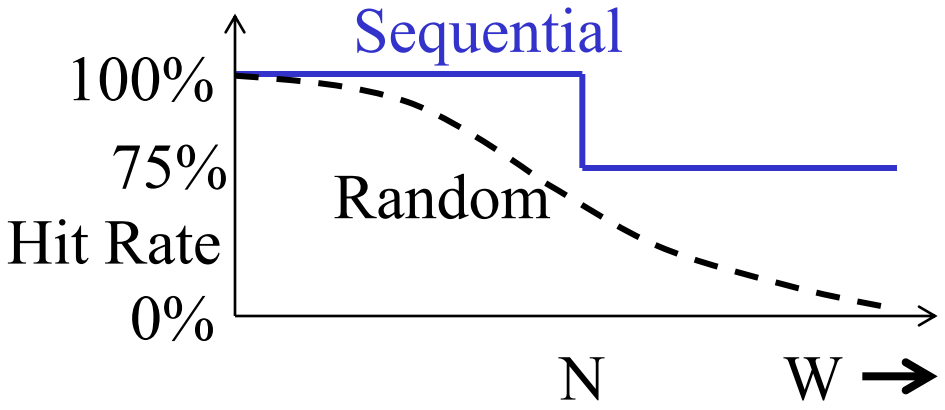
$$\frac{N-1}{N}$$

Case study in cache hit rate vs locality

- Consider simplest possible regular memory access pattern:
  - Working set size (number of locations used) = W
  - Sequential circular access:
    - A, A+1,A+2, ..., B-1, B, A,
    - Sequential access from A to B wrapping round back to A.
    - NB these are word numbers (multiply by 4 for 32 bit ARM)
  - Here W = B - A+1
- Direct mapped cache size N, W < N => 100% hit rate
- Compare block size 1 (no locality) with block size 4 (4 contiguous words fetched whenever one word in block is accessed).

- **Block size = 1**
  - No spatial locality
  - Temporal locality useful if N ≤ W
- **Block size = 4**
  - Spatial locality gives 75% hit rate
  - Temporal locality as before
- Note that this access pattern (sequential access) is optimal for spatial locality
- Random access (dotted line) has no spatial locality.





Direct Mapped Cache – Key points

- Caches exploit *temporal* & *spatial* locality to get high hit rate
- A particular memory item is stored in a unique location in the cache.
- To check if a particular memory item is in cache, the **index** bits of the memory address are used to address the cache entry.
- The top memory address bits are then compared with the stored **tag**. If they are equal, and the V bit is 1, we have got a hit.
- The bottom memory address bits **select** determine byte within cache line
- **Tag** is all address bits except **index** and **select**.
  - Size of cache with N bits index and M bits select is $2^{(N+M)}$ bytes
- When a miss occurs, data cannot be read from the cache. A slower read from the next level of memory must take place, incurring a miss penalty.
- Note V bit is only necessary to mark initial state where cache contains no valid data (hence must always be a miss)