K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCEAND ENGINEERING

Course File

BCS402 – Microcontrollers

IV SEM 'A' Sec (2023-24)

Faculty In-charge
Mrs. Meena G
Assistant Professor
Department of Computer Science and Engineering
K S School of Engineering & Management, Bangalore

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

CONTENTS

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VISION

"To produce quality Computer Science professional, possessing excellent technical knowledge, skills, personality through education and research."

MISSION

Department of Computer Science and Engineering shall,

M1: Provide good infrastructure and facilitate learning to become competent engineers who meet global challenges.

M2: Encourages industry institute interaction to give an edge to the students.

M3: Facilitates experimental learning through interdisciplinary projects.

M4: Strengthen soft skill to address global challenges.

Program Educational Objectives (PEOs)

Graduates of B.E. program in Computer Science and Engineering will be able to:

PEO1: Analyze Design, Simulate and Solve engineering problems to become an efficient Software Engineer in diverse fields and will be a successful individual.

PEO2: Work effectively and efficiently as individual and/or in a team, exhibiting leadership qualities with strong communicational skills along with professional and ethical values.

PEO3: Become an entrepreneur/inventor to design and develop product/system to meet social, technical, environmental and business needs.

PEO4: Engage in learning leading to higher education and research.

Program Specific Outcomes (PSOs)

PSO1: Understand fundamental and advanced concepts in the core areas of Computer Science and Engineering to analyze, design and implement the solutions for the real world problems.

PSO2: Utilize modern technological innovations efficiently in various applications to work towards the betterment of society and solve engineering problems.

MICROCONTROLLERS		Semester	4
Course Code	BCS402	CIE Marks	50
Teaching Hours/Week (L:T:P: S)	3:0:2:0	SEE Marks	50
Total Hours of Pedagogy	40 hours Theory + 8-10 Lab Slots	Total Marks	100
Credits	04	Exam Hours	3
Examination nature (SEE)	Theory		

Course Objectives:

CLO 1: Understand the fundamentals of ARM-based systems and basic architecture of CISC and RISC.
CLO 2: Familiarize with ARM programming modules along with registers, CPSR and Flags.
CLO 3: Develop ALP using various instructions to program the ARM controller.
CLO 4: Understand the Exceptions and Interrupt handling mechanism in Microcontrollers.
CLO 5: Discuss the ARM Firmware packages and Cache memory polices.

Teaching-Learning Process

These are sample Strategies, which teachers can use to accelerate the attainment of the various course outcomes.

1. Lecturer method (L) needs not to be only a traditional lecture method, but alternative effective teaching methods could be adopted to attain the outcomes.
2. Use of Video/Animation to explain functioning of various concepts.
3. Encourage collaborative (Group Learning) Learning in the class.
4. Ask at least three HOT (Higher order Thinking) questions in the class, which promotes critical thinking.
5. Adopt Problem Based Learning (PBL), which fosters students' Analytical skills, develop design thinking skills such as the ability to design, evaluate, generalize, and analyze information rather than simply recall it.
6. Introduce Topics in manifold representations.
7. Show the different ways to solve the same problem with different circuits/logic and encourage the students to come up with their own creative ways to solve them.
8. Discuss how every concept can be applied to the real world - and when that's possible, it helps improve the students understanding.
9. Use any of these methods: Chalk and board, Active Learning, Case Studies.

MODULE-1	No. of Hours: 8

ARM Embedded Systems: The RISC design philosophy, The ARM Design Philosophy, Embedded System Hardware, Embedded System Software.

ARM Processor Fundamentals: Registers, Current Program Status Register, Pipeline, Exceptions, Interrupts, and the Vector Table, Core Extensions

Textbook 1: Chapter 1 - 1.1 to 1.4, Chapter 2 - 2.1 to 2.5
RBT: L1, L2, L3

MODULE-2	No. of Hours: 8

Introduction to the ARM Instruction Set: Data Processing Instructions, Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants.

Textbook 1: Chapter 3 - 3.1 to 3.6
RBT: L1, L2, L3

MODULE-3	No. of Hours:8

C Compilers and Optimization: Basic C Data Types, C Looping Structures, Register Allocation, Function Calls, Pointer Aliasing, Portability Issues.

Textbook 1: Chapter 5.1 to 5.7 and 5.13
RBT: L1, L2, L3

MODULE-4	No. of Hours:8
Exception and Interrupt Handling: Exception handling, ARM processor exceptions and modes, vector table, exception priorities, link register offsets, interrupts, assigning interrupts, interrupt latency, IRQ and FIQ exceptions, basic interrupt stack design and implementation. **Firmware:** Firmware and bootloader, ARM firmware suite, Red Hat redboot, Example: sandstone, sandstone directory layout, sandstone code structure. **Textbook 1: Chapter 9.1 and 9.2, Chapter 10** **RBT: L1, L2, L3**	
MODULE-5	**No. of Hours:08**
CACHES: The Memory Hierarchy and Cache Memory, Caches and Memory Management Units: CACHE Architecture: Basic Architecture of a Cache Memory, Basic Operation of a Cache Controller, The Relationship between Cache and Main Memory, Set Associativity, Write Buffers, Measuring Cache Efficiency, CACHE POLICY: Write Policy—Writeback or Writethrough, Cache Line Replacement Policies, Allocation Policy on a Cache Miss. Coprocessor 15 and caches. **Textbook 1: Chapter 12.1 to 12.4** **RBT: L1, L2, L3**	

PRACTICAL COMPONENT OF IPCC *(May cover all / major modules)*

Sl.No.	Experiments
Module – 1	
1.	Using Keil software, observe the various Registers, Dump, CPSR, with a simple Assembly Language Programs (ALP).
Module – 2	
2.	Develop and simulate ARM ALP for Data Transfer, Arithmetic and Logical operations (Demonstrate with the help of a suitable program).
3.	Develop an ALP to multiply two 16-bit binary numbers.
4.	Develop an ALP to find the sum of first 10 integer numbers.
5.	Develop an ALP to find the largest/smallest number in an array of 32 numbers.
6.	Develop an ALP to count the number of ones and zeros in two consecutive memory locations.
Module – 3	
7.	Simulate a program in C for ARM microcontroller using KEIL to sort the numbers in ascending/descending order using bubble sort.
8.	Simulate a program in C for ARM microcontroller to find factorial of a number.
9.	Simulate a program in C for ARM microcontroller to demonstrate case conversion of characters from upper to lowercase and lower to uppercase.
Module – 4 and 5	
10.	Demonstrate enabling and disabling of Interrupts in ARM.
11.	Demonstrate the handling of divide by zero, Invalid Operation and Overflow exceptions in ARM.

Course outcomes (Course Skill Set):

At the end of the course, the student will be able to:

- Explain the ARM Architectural features and Instructions.
- Develop programs using ARM instruction set for an ARM Microcontroller.
- Explain C-Compiler Optimizations and portability issues in ARM Microcontroller.
- Apply the concepts of Exceptions and Interrupt handling mechanisms in developing applications.
- Demonstrate the role of Cache management and Firmware in Microcontrollers.

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the

academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks.**
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks).**
- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

1. **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
2. On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
3. The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks.**
4. The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
5. Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks.**
6. The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks.

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component.

Suggested Learning Resources:

Text Books:

1. Andrew N Sloss, Dominic Symes and Chris Wright, ARM system developers guide, Elsevier, Morgan Kaufman publishers, 2008.

Reference Books:

1. Raghunandan.G.H, Microcontroller (ARM) and Embedded System, Cengage learning Publication, 2019.
2. Insider's Guide to the ARM7 based microcontrollers, Hitex Ltd.,1st edition, 2005

Activity Based Learning (Suggested Activities in Class)/ Practical Based Learning

Assign the group task to demonstrate the Installation and working of Keil Software.

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT
BENGALURU-560109

TENTATIVE CALENDAR OF EVENTS: IV EVEN SEMESTER (2023-2024)

SESSION: APRIL TO AUG 2024

Week No.	Month	Day						Days	Activities
		Mon	Tue	Wed	Thu	Fri	Sat		
1	APR	22*	23	24	25	26 H	27	5	22* - Commencement of IV sem 26- Election 27- Wednesday Time Table
2	APR/ MAY	29	30	1 H	2	3	4 DH	4	1- May Day
3	MAY	6	7	8	9	10 H	11	5	10 - Basava Jayanthi 11- Friday Time Table
4	MAY	13	14	15	16	17 TA	18 DH	5	
5	MAY	20 T1	21 T1	22 T1	23	24	25	6	25- Monday Time Table
6	MAY/ JUNE	27 BV	28 ASD	29* FFB1	30	31	1 DH	5	29 - First Faculty Feed Back
7	JUNE	3	4	5	6	7	8	6	8- Monday Time Table
8	JUNE	10	11	12	13	14	15 DH	5	
9	JUNE	17 H	18	19	20	21	22 TA	5	17- Bakrid 22- Wednesday Time Table
10	JUNE	24T2	25 T2	26 T2	27	28	29	6	29- Tuesday Time Table
11	JULY	1BV	2 ASD	3* FFB2	4	5	6 DH	5	3 - First Faculty Feed Back
12	JULY	8	9	10	11	12	13	6	13- Friday Time Table
13	JULY	15	16	17 H	18	19T A	20 DH	4	17- Last Day of Moharam
14	JULY	22	23	24	25	26	27	6	27- Wednesday Time Table
15	JULY / AUG	29T3	30T3	31 T3	1 LT	2 LT	3D H	5	
16	AUG	5 LT	6	7*				3	**7* - Last Working day**
Total No of Working Days : 81									

Total Number of working days (Excluding holidays and Tests)=69

H	Holiday
BV	Blue Book Verification
T1,T2,T3	Tests 1,2,3
ASD	Attendance & Sessional Display
DH	Declared Holiday
LT	Lab Test
TA	Test attendance

Monday	13
Tuesday	14
Wednesday	14
Thursday	14
Friday	14
Total	**69**

SIGNATURE OF PRINCIPAL
Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

CLASS TIME TABLE

(w.e.f. 22/4/2024)

Class: IV CSE 'A' Lecture Hall: A404 Class Teacher: Mrs. Meena G

DAY	8.40-9.35	9.35-10.30	10.30-10.45	10.45-11.40	11.40-12.35	12.35-1.20	1.20-2.10	2.10-3.00	3.00-3.50
MONDAY	Microcontrollers Lab Batch - A1 Analysis & Design of Algorithms Lab Batch - A2				DMS (BCS405A)	LUNCH BREAK	DBMS (BCS403)	MC (BCS402)	Library
TUESDAY	Microcontrollers Lab Batch -A2 Analysis & Design of Algorithms Lab Batch - A1				ADA (BCS401)		DMS (BCS405A)	MC (BCS402)	DBMS (BCS403)
WEDNESDAY	DBMS (BCS403)	ADA (BCS401)	TEA BREAK	DMS (BCS405A)	Bio (BBOK407)		Database Management Systems Lab Batch - A1 Technical writing using LATEX Lab Batch -A2		
THURSDAY	DMS (BCS405A)	ADA (BCS401)		DBMS (BCS403)	MC (BCS402)		Database Management Systems Lab Batch - A2 Technical writing using LATEX Lab Batch -A1		
FRIDAY	UHV (BUHK408)	ADA (BCS401)		Bio (BBOK407)	MC (BCS402)		Tutorial	NSS / Yoga / Sports	
SATURDAY	AS PER CALENDAR OF EVENTS								

SUBJECT CODE	SUBJECT NAME	FACULTY NAME
BCS401	Analysis & Design of Algorithms	Dr K Venkata Rao
BCS402	Microcontrollers	Mrs Meena G
BCS402	Microcontrollers Lab (BCS402)	Mrs Meena G Mrs Sushmitha Suresh
BCS403	Database Management Systems	Mrs Sougandhika Narayan
BCS403	Database Management Systems Lab	Mrs Sougandhika Narayan Mrs Dakshayini G R
BCSL404	Analysis & Design of Algorithms Lab	Mrs Kavitha K S Mrs N Deepashree
BCS405A	Discrete Mathematical Structure (BCS405A)	Mr Mohnar
BCS456D	Technical writing using LATEX (Lab)	Mrs Nagaveni B Nimbal Mrs R S Geethanjali
BBOK407	Biology For Engineers	Ms Rashmi
BUHK408	Universal human values course	Mrs Dakshayini G R
BNSK459	National Service Scheme (NSS)	Mrs Prasanna N Mrs Dakshayini G R
BPEK459	Physical Education (PE) (Sports and Athletics)	Mrs Kavitha K S Mrs Meena G
BYOK459	Yoga	Mrs Bindu K P
	Library	Mrs Sougandhika Narayan

Time Table Coordinator

Head of the Department
Computer Science & Engineering
K.S School of Engineering & Management
Bangaloro-560109

Dr. K. RAMANARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

SESSION: 2023-2024(EVEN SEMESTER)

(w. e. f : 22/4/2024)

INDIVIDUAL TIME TABLE

Class: IV 'A & B' **Faculty Name: Mrs. Meena G**

DAY	8.40-9.35	9.35-10.30	10.30 -10.45	10.45 -11.40	11.40-12.35	12.35-1.20	1.20 -2.10	2.10-3.00	3.00-3.50
MONDAY	Microcontrollers Lab Batch -A1					LUNCH BREAK	Project Work Phase II (G1/G4)	MC (IV A)	Project Work Phase II (G1/G4)
TUESDAY	Microcontrollers Lab Batch -A2						Project Work Phase II (G2/G5)	MC (IV A)	Project Work Phase II (G2/G5)
WEDNESDAY	Microcontrollers Lab Batch - B1				MC (IV A)		Project Work Phase II(G3/G6) (18CSP83)		
THURSDAY	Microcontrollers Lab Batch - B2				MC (IV A)				
FRIDAY								Sporst (IV A)	
SATURDAY	AS PER CALENDAR OF EVENTS								

CODE	SUBJECT	Hours /Week
BCS402	Microcontrollers	4
BCS402	Microcontrollers Lab	6
BPEK459	Physical Education (PE) (Sports and Athletics)	2
18CSI85	Internship	1
18CSS84	Technical Seminar	1
18CSP83	Project Work Phase-II	4.5

Mrs. Meena G

Time-table Coordinator

HOD
Department of Computer Science Engineering
K.S School of Engineering and Management
Bangalore-560109
Head of the Department

Dr. K RAMA NARASIMHA
Principal/Director
K S School of Engineering and Management
Bengaluru - 560 109
Principal

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU -560 019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

IV Semester - A Student List

Sl.	USN	Name of the Student
1	1KG22CS001	A G VISHNU
2	1KG22CS002	A PAVITHRA
3	1KG22CS003	ABHISHEK S
4	1KG22CS004	AKSHAYA K
5	1KG22CS005	ALLURU VENKATASAI JYOTHISHREDDY
6	1KG22CS006	AMISHA V
7	1KG22CS007	ANAGHA S
8	1KG22CS008	ANANTHANENI KRISHNA SAI
9	1KG22CS009	ANCHAL R S SINGH
10	1KG22CS010	ANJANA R C
11	1KG22CS011	AVINASH NAYAK M
12	1KG22CS012	B M DARSHAN
13	1KG22CS013	B N RUSHITHA
14	1KG22CS014	B USHASREE
15	1KG22CS015	B V DEEKSHA JAIN
16	1KG22CS016	BHANU PRIYA K
17	1KG22CS017	BHARATH R N
18	1KG22CS018	BHAVYA D
19	1KG22CS019	BHEEMANNA
20	1KG22CS020	BOURISETTI CHAITANYA
21	1KG22CS021	BYNI PURUSHOTHAM
22	1KG22CS022	C GOWTHAM
23	1KG22CS023	C R ANAGHA
24	1KG22CS024	CHAITHANYA C
25	1KG22CS025	CHAITRA C
26	1KG22CS026	CHALLA PAVAN KUMAR
27	1KG22CS027	CHARAN KUMAR P K
28	1KG22CS028	CHARAN T R
29	1KG22CS029	D MAHESH
30	1KG22CS030	D SHREYAS
31	1KG22CS031	DANDA SHALINI
32	1KG22CS032	DASARI YASASWI NANDA
33	1KG22CS033	DEEKSHA D SHENOY
34	1KG22CS034	DISHA S
35	1KG22CS035	DIYA AJITH KASABEKAR
36	1KG22CS036	E MADAN KUMAR
37	1KG22CS037	ENTURI LOKESH
38	1KG22CS038	G SHARATH RAJ
39	1KG22CS039	G UHA
40	1KG22CS040	GAGANA SHREE S
41	1KG22CS041	GANASHREE C N
42	1KG22CS042	GOLLA KAVYA
43	1KG22CS043	GOLLA KUSUMA
44	1KG22CS044	GONUGUNTLA PRASHANTH KUMAR

45	1KG22CS045	GORTHI YASWANTH
46	1KG22CS046	HARISH R A
47	1KG22CS047	HARSHA C V
48	1KG22CS048	HARSHITHA C K
49	1KG22CS049	HEMANTH R
50	1KG22CS050	HRISHIKESH B S
51	1KG22CS051	JAJAPPAGARI SAI SREE
52	1KG22CS052	JAMPULA ABHILASH
53	1KG22CS053	K GAYATHRI
54	1KG22CS054	K PRAMOD KUMAR
55	1KG22CS055	KAVANA S M
56	1KG22CS056	KAVYA S
57	1KG22CS057	KEERTHANA B
58	1KG22CS058	KOLLA BHAVANA
59	1KG22CS059	KOUSIK N
60	1KG22CS060	L R DHAYATRI
61	1KG22CS061	LIKHITHA P V
62	1KG22CS062	LIKITHA R
63	1KG22CS063	M SAIJA
64	1KG23CS400	GOWTHAM TM
65	1KG23CS401	IQRAA IMAN KHAN
66	1KG23CS402	MUDDASSIR AHMED I TORGUL

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU -560 019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

IV Semester - A Batch List

Sl. No.	USN	Name of the Student	Batch
1	1KG22CS001	A G VISHNU	
2	1KG22CS002	A PAVITHRA	
3	1KG22CS003	ABHISHEK S	
4	1KG22CS004	AKSHAYA K	
5	1KG22CS005	ALLURU VENKATASAI JYOTHISHREDDY	
6	1KG22CS006	AMISHA V	
7	1KG22CS007	ANAGHA S	
8	1KG22CS008	ANANTHANENI KRISHNA SAI	
9	1KG22CS009	ANCHAL R S SINGH	
10	1KG22CS010	ANJANA R C	
11	1KG22CS011	AVINASH NAYAK M	
12	1KG22CS012	B M DARSHAN	
13	1KG22CS013	B N RUSHITHA	
14	1KG22CS014	B USHASREE	
15	1KG22CS015	B V DEEKSHA JAIN	
16	1KG22CS016	BHANU PRIYA K	
17	1KG22CS017	BHARATH R N	**Batch A1**
18	1KG22CS018	BHAVYA D	
19	1KG22CS019	BHEEMANNA	
20	1KG22CS020	BOURISETTI CHAITANYA	
21	1KG22CS021	BYNI PURUSHOTHAM	
22	1KG22CS022	C GOWTHAM	
23	1KG22CS023	C R ANAGHA	
24	1KG22CS024	CHAITHANYA C	
25	1KG22CS025	CHAITRA C	
26	1KG22CS026	CHALLA PAVAN KUMAR	

27	1KG22CS027	CHARAN KUMAR P K	
28	1KG22CS028	CHARAN T R	
29	1KG22CS029	D MAHESH	
30	1KG22CS030	D SHREYAS	
31	1KG22CS031	DANDA SHALINI	
32	1KG22CS032	DASARI YASASWI NANDA	
33	1KG22CS033	DEEKSHA D SHENOY	
34	1KG22CS034	DISHA S	
35	1KG22CS035	DIYA AJITH KASABEKAR	
36	1KG22CS036	E MADAN KUMAR	
37	1KG22CS037	ENTURI LOKESH	
38	1KG22CS038	G SHARATH RAJ	
39	1KG22CS039	G UHA	
40	1KG22CS040	GAGANA SHREE S	
41	1KG22CS041	GANASHREE C N	
42	1KG22CS042	GOLLA KAVYA	
43	1KG22CS043	GOLLA KUSUMA	
44	1KG22CS044	GONUGUNTLA PRASHANTH KUMAR	
45	1KG22CS045	GORTHI YASWANTH	
46	1KG22CS046	HARISH R A	
47	1KG22CS047	HARSHA C V	
48	1KG22CS048	HARSHITHA C K	
49	1KG22CS049	HEMANTH R	
50	1KG22CS050	HRISHIKESH B S	**Batch A2**
51	1KG22CS051	JAJAPPAGARI SAI SREE	
52	1KG22CS052	JAMPULA ABHILASH	
53	1KG22CS053	K GAYATHRI	
54	1KG22CS054	K PRAMOD KUMAR	
55	1KG22CS055	KAVANA S M	
56	1KG22CS056	KAVYA S	
57	1KG22CS057	KEERTHANA B	

58	1KG22CS058	KOLLA BHAVANA
59	1KG22CS059	KOUSIK N
60	1KG22CS060	L R DHAYATRI
61	1KG22CS061	LIKHITHA P V
62	1KG22CS062	LIKITHA R
63	1KG22CS063	M SAIJA
64	1KG23CS400	GOWTHAM TM
65	1KG23CS401	IQRAA IMAN KHAN
66	1KG23CS402	MUDDASSIR AHMED I TORGUL

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE& ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

LESSON PLAN

NAME OF THE STAFF : **Meena G**

COURSE CODE/TITLE : **BCS402/MICROCONTROLLER**

SEMESTER/YEAR : **IV/II – A Section**

Sl. No.	Topic to be covered	Mode of Delivery	Teaching Aid	No. of Periods	Cumulative No. of Periods	Proposed Date	Delivery Date
	MODULE 1						
1	**ARM Embedded Systems:** The RISC design philosophy	L+D	BB+LCD	1	1	22/4/2024	24|4|24
2	The ARM Design Philosophy	L+D	BB+LCD	1	2	23/4/2024	25|4|24
3	Embedded System Hardware	L+D	BB+LCD	1	3	25/4/2024	29|4|24
4	Embedded System Software	L+D	BB+LCD	1	4	29/4/2024	30|4|24
5	**ARM Processor Fundamentals:** Registers	L+D	BB+LCD	1	5	30/4/2024	02|5|24
6	Current Program Status Register	L+D	BB+LCD	1	6	02/05/2024	15|5|24
7	Exceptions, Interrupts, Pipeline	L+D	BB+LCD	1	7	03/05/2024	16|5|24
8	The Vector Table , Core Extension	L+D	BB+LCD	1	8	06/05/2024	17|5|24
10	Tutorial	L+D	BB+LCD	4	-	07/05/2024 09/05/2024	20|5|24
	MODULE 2						
11	**Introduction to the ARM Instruction Set:** Data Processing Instructions	L+D	BB+LCD	1	9	11/05/2024	20|5|24

12	Data Processing Instructions(continued)	L+D	BB+LCD	1	10	13/05/2024	21|5|24
13	Branch Instructions	L+D	BB+LCD	1	11	14/05/2024	22|5|24
14	Branch Instructions(continued)	L+D	BB+LCD	1	12	16/05/2024	23|5|24
15	Software Interrupt Instructions	L+D	BB+LCD	1	13	17/05/2024	23|5|24
16	Program Status Register Instructions	L+D	BB+LCD	1	14	23/05/2024	24|5|24
17	Coprocessor Instructions	L+D	BB+LCD	1	15	24/05/2024	24|5|24
18	Loading Constants	L+D	BB+LCD	1	16	25/05/2024	25|5|24
20	Tutorial	L+D	BB+LCD	4	-	27/05/2024 28/05/2024	27|5|24
	MODULE 3						
21	**C Compilers and Optimization:** Basic C Data Types	L+D	BB+LCD	1	17	30/05/2024	6|6|24
22	C Looping Structures	L+D	BB+LCD	1	18	31/05/2024	7|6|24
23	C Looping Structures(continued)	L+D	BB+LCD	1	19	03/06/2024	25|6|24
24	Register Allocation	L+D	BB+LCD	1	20	04/06/2024	25|6|24
25	Register Allocation(continued)	L+D	BB+LCD	1	21	06/06/2024	27|6|24
26	Function Calls	L+D	BB+LCD	1	22	07/06/2024	28|6|24
27	Pointer Aliasing	L+D	BB+LCD	1	23	08/06/2024	29|6|24
28	Portability Issues	L+D	BB+LCD	1	24	10/06/2024	1|7|24
30	Tutorial	L+D	BB+LCD	4	-	11/06/2024 13/06/2024	1|7|24
	MODULE 4						
31	**Exception and Interrupt Handling:** Exception handling, ARM processor exceptions and modes	L+D	BB+LCD	1	25	14/06/2024	2|7|24
32	Vector table, exception priorities, link register offsets	L+D	BB+LCD	1	26	18/06/2024	3|7|24
33	Interrupts, assigning interrupts,	L+D	BB+LCD	1	27	20/06/2024	4|7|24

	interrupt latency, IRQ and FIQ exceptions						
34	Firmware and bootloader	L+D	BB+LCD	1	28	21/06/2024	4|7|24
35	ARM firmware suite	L+D	BB+LCD	1	29	27/06/2024	5|7|24
36	Red Hat redboot	L+D	BB+LCD	1	30	28/06/2024	12|7|24
37	Example:sandstone,sandstone directory layout	L+D	BB+LCD	1	31	29/06/2024	13|7|24
38	Sandstone code structure.	L+D	BB+LCD	1	32	01/07/2024	15|7|24
40	Tutorial	L+D	BB+LCD	4	-	02/07/2024 04/07/2024	15|7|24
	MODULE 5						
41	**CACHES:** The Memory Hierarchy and Cache Memory	L+D	BB+LCD	1	33	05/07/2024	16|7|24
42	Caches and Memory Management Units	L+D	BB+LCD	1	34	08/07/2024	18|7|24
43	CACHE Architecture: Basic Architecture of a Cache Memory, Basic Operation of a Cache Controller	L+D	BB+LCD	1	35	08/07/2024	19|7|24
44	The Relationship between Cache and Main Memory, Set Associativity	L+D	BB+LCD	1	36	09/07/2024	23|7|24
45	Write Buffers, Measuring Cache Efficiency	L+D	BB+LCD	1	37	11/07/2024	23|7|24
46	CACHE POLICY: Write Policy—Writeback or Writethrough	L+D	BB+LCD	1	38	12/07/2024	25|7|24
47	Cache Line Replacement Policies	L+D	BB+LCD	1	39	13/07/2024	25|7|24
48	Allocation Policy on a Cache Miss, Coprocessor 15 and caches	L+D	BB+LCD	1	40	15/07/2024	26|7|24
50	Tutorial	L+D	BB+LCD	4	-	16/07/2024	27|7|24

51	Revision	L+D	BB+LCD	6	-	18/07/2024 - 07/08/2024	—
52	**PRATICALS:** 1. Using Keil software, observe the various Registers, Dump, CPSR, with a simple Assembly Language Programs (ALP).	Practical	D	3	3	A1:22/04/2024 A2:23/04/2024	A1:22|4|24 A2:23|4|24
	2. Develop and simulate ARM ALP for Data Transfer, Arithmetic and Logical operations (Demonstrate with the help of a suitable program).	Practical	D	3	6	A1:29/04/2024 A2:30/04/2024	A1:29|4|24 A2:30|4|24
	3. Develop an ALP to multiply two 16-bit binary numbers.	Practical	D	3	9	A1:06/05/2024 A2:07/05/2024	A1:29|4|24 A2:14|5|24
	4. Develop an ALP to find the sum of first 10 integer numbers.	Practical	D	3	12	A1:13/05/2024 A2:14/05/2024	A1:13|5|24 A2:14|5|24
	5. Develop an ALP to find the largest/smallest number in an array of 32 numbers.	Practical	D	3	15	A1:27/05/2024 A2:28/05/2024	A1:13|5|24 A2:21|5|24
	6. Develop an ALP to count the number of ones and zeros in two consecutive memory locations.	Practical	D	3	18	A1:03/06/2024 A2:04/06/2024	A1:8|6|24 A2:4|6|24
	7. Simulate a program in C for ARM microcontroller using KEIL to sort the numbers in ascending/descending order using bubble sort.	Practical	D	3	21	A1:10/06/2024 A2:11/06/2024	A1:24|6|24 A2:21|6|24
	8. Simulate a program in C for ARM microcontroller to find factorial of a number.	Practical	D	3	24	A1:01/07/2024 A2:18/06/2024	A1:1|7|24 A2:2|7|24

	9. Simulate a program in C for ARM microcontroller to demonstrate case conversion of characters from upper to lowercase and lower to uppercase.	Practical	D	3	27	A1:08/07/2024 A2:02/07/2024	A1:15/7/24 A2:13/7/24
	10. Demonstrate enabling and disabling of Interrupts in ARM.	Practical	D	3	30	A1:15/07/2024 A2:09/07/2024	A1:22/7/24 A2:16/7/24
	11. Demonstrate the handling of divide by zero, Invalid Operation and Overflow exceptions in ARM.	Practical	D	3	33	A1:22/07/2024 A2:16/07/2024	A1:29/7/24 A2:23/7/24
	12. Practical Revision	Practical	D	3	-	A1:05/08/2024 A2:23/07/2024 A2:06/08/2024	A1: — A2-30/7/24

	Week	Remarks
Assignment 1	**4th Week – 14/05/2024**	**Mode of Assignment – Written Assignment**
Assignment 2	**9th Week- 18/07/2024**	

Total No. of Lecture Hours = 40
Total No. of Tutorial Hours =26
Total No. of Practical Hours = 20

Course In charge

Head of Dept

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

IQAC Coordinator

Principal

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Manage
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

QUESTION BANK - 1

Microcontroller (BCS402)

Module-1

1. **Explain** RISC design philosophy.
2. **Explain** ARM design philosophy.
3. With neat sketch, **Explain** ARM processor based embedded system hardware.
4. **Explain** ARM processor based embedded system software.
5. **Discuss** ARM core dataflow model
6. **Explain** active registers available in user mode.
7. **Explain** current program status register (cpsr)
8. **Explain** processor modes for ARM processor.
9. **Summarize** complete ARM register set.
10. **Discuss** pipeline concept of ARM processor.
11. **Write a note** on exception, interrupt and the vector table.
12. **Discuss** the core extensions for ARM processor
13. **Differentiate** Microprocessor and Microcontroller with diagram.
14. **Differentiate** CISC and RISC processors
15. **Differentiate** Harvard and Von-Neumann architecture.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

QUESTION BANK - 2

Microcontroller (BCS402)

Module-2

1. **Determine** the operation of data processing instructions of ARM with an example.
2. **Illustrate** the working of Barrel shifter with example.
3. **Demonstrate** the working of Branch Instructions with example.
4. **Identify** the working of Load and Store instructions with examples.
5. **Make use of** software interrupt instruction to solve the given problem

```
PRE     cpsr = nzcVqifT_USER
        pc = 0x00008000
        lr = 0x003fffff          ; lr = r14
        r0 = 0x12

    0x00008000    SWI    0x45
```

6. **Determine** the operation of program status register instructions with an example.
7. **Illustrate** coprocessor instructions with syntax and example.
8. **Interpret** loading constants with syntax.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

QUESTION BANK - 3

Microcontroller (BCS402)

Module-3

1. **Illustrate** the importance of basic C data types with an example.
2. **Interpret C** looping structure with an example.
3. **Write a note** on register allocation.
4. **Write a note** function call.
5. **Illustrate** pointer aliasing with an example
6. **Interpret** structure arrangement with an example.
7. **Discuss** various Portability issues in detail.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

QUESTION BANK - 4

Microcontroller (BCS402)

Module-4

1. **Define** exception and interrupt. **Explain** different types of exception.
2. **Interpret** the concept of exception handling.
3. **Outline** the following
   i. Assigning interrupts
   ii. Interrupt latency
   iii. IRQ and FIQ exceptions
   iv. Basic interrupt stack design and implementation
4. **Define** firmware. **Explain** firmware execution flow in detail.
5. **Discuss** various firmware suites in detail.
6. **Interpret** Sandstone Directory Layout.
7. **Illustrate** Sandstone Code Structure/ Sandstone execution flow.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

QUESTION BANK - 5

Microcontroller (BCS402)

Module-5

1. **Define** cache. **Explain** memory hierarchy and cache memory.
2. **Outline** the following
   i. Relationship that cache has between processor core and main memory
   ii. Caches and memory management units
3. **Discuss** cache architecture.
4. **Interpret** the following
   i. Basic architecture of a cache memory
   ii. Basic operation of a cache controller
   iii. The relationship between cache and main memory
5. **Explain** the process of thrashing with suitable diagram.
6. **Illustrate** set associativity.
7. **Summarize** the following
   i. Write buffers
   ii. Measuring cache efficiency
   iii. Coprocessor 15 and caches
8. **Explain** in detail various cache policies.

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU - 560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION : 2023-24 [EVEN SEM]

CO-PO Mapping

Course: Microcontroller				
Type: Integrated Professional Core Course			Course Code: BCS402	
No of Hours				
Theory (Lecture Class)	Tutorials	Practical/Field Work/Allied Activities	Total/Week	Total hours of Pedagogy
3	0	2	5	40 T + 20 P
Marks				
CIE	SEE		Total	Credits
50	50		100	4

Aim/Objectives of the Course

1. Understand the fundamentals of ARM-based systems and basic architecture of CISC and RISC.
2. Familiarize with ARM programming modules along with registers, CPSR and Flags.
3. Develop ALP using various instructions to program the ARM controller.
4. Understand the Exceptions and Interrupt handling mechanism in Microcontrollers.
5. Discuss the ARM Firmware packages and Cache memory polices.

Course Learning Outcomes

After completing the course, the students will be able to

CO1	**Explain** the ARM Architectural features and Instructions.	**Applying (K3)**
CO2	**Develop** programs using ARM instruction set for an ARM Microcontroller	**Applying (K3)**
CO3	**Explain** C-Compiler Optimizations and portability issues in ARM Microcontroller	**Applying (K3)**
CO4	**Apply** the concepts of Exceptions and Interrupt handling mechanisms in developing applications.	**Applying (K3)**
CO5	**Demonstrate** the role of Cache management and Firmware in Microcontrollers.	**Applying (K3)**

Syllabus Content

Module 1:ARM Embedded Systems: The RISC design philosophy, The ARM Design Philosophy, Embedded System Hardware, Embedded System Software. **ARM Processor Fundamentals:** Registers, Current Program Status Register, Pipeline, Exceptions, Interrupts, and the Vector Table, Core Extensions **LO:** At the end of this session the student will be able to 1. Understand the design philosophy of ARM processor. 2. Understand the fundamentals of ARM processor	**CO1** 8hrs PO1-3 PO2-2 PO3-2 PO5-3 PO6-2 PO12-2

	PSO1-3 PSO2-2
Module 2: Introduction to the ARM Instruction Set: Data Processing Instructions, Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants. **LO:**At the end of this session the student will be able to 1. Understand the different instructions and registers of ARM processor. 2. Demonstrate the programming using ARM instructions.	**CO2** 8 hrs. PO1-3 PO2-2 PO3-3 PO5-3 PO6-2 PO12-2 PSO1-3 PSO2-2
Module 3: C Compilers and Optimization: Basic C Data Types, C Looping Structures, Register Allocation, Function Calls, Pointer Aliasing, Portability Issues. **LO:** At the end of this session the student will be able to 1. Explain the operation of assembly code and their programs. 2. Understand the execution of Assembly language programs.	**CO3** 8hrs PO1-3 PO2-3 PO3-2 PO5-3 PO6-2 PO12-2 PSO1-3 PSO2-2
Module 4:Exception and Interrupt Handling: Exception handling, ARM processor exceptions and modes, vector table, exception priorities, link register offsets, interrupts, assigning interrupts, interrupt latency, IRQ and FIQ exceptions, basic interrupt stack design and implementation. **Firmware:** Firmware and bootloader, ARM firmware suite, Red Hat redboot, Example: sandstone, sandstone directory layout, sandstone code structure. **LO:** At the end of this session the student will be able to 1. Understand the concept exception handling. 2. Explain about ARM processor exceptions and modes. 3. Understand the working of firmware.	**CO4** 8 hrs PO1-3 PO2-3 PO3-3 PO5-3 PO6-2 PO12-2 PSO1-3 PSO2-2
Module 5:CACHES: The Memory Hierarchy and Cache Memory, Caches and Memory Management Units: CACHE Architecture: Basic Architecture of a Cache Memory, Basic Operation of a Cache Controller, The Relationship between Cache and Main Memory, Set Associativity, Write Buffers, Measuring Cache Efficiency, CACHE POLICY: Write Policy—Writeback or Writethrough, Cache Line Replacement Policies, Allocation Policy on a Cache Miss. Coprocessor 15 and caches	**CO5** 8 hrs PO1-3 PO2-3

- The student has to secure 40% of 25 marks to qualify in the CIE of the theory component of IPCC.

CIE for the practical component of the IPCC

1. **15 marks** for the conduction of the experiment and preparation of laboratory record, and **10 marks** for the test to be conducted after the completion of all the laboratory sessions.
2. On completion of every experiment/program in the laboratory, the students shall be evaluated including viva-voce and marks shall be awarded on the same day.
3. The CIE marks awarded in the case of the Practical component shall be based on the continuous evaluation of the laboratory report. Each experiment report can be evaluated for 10 marks. Marks of all experiments' write-ups are added and scaled down to **15 marks**.
4. The laboratory test **(duration 02/03 hours)** after completion of all the experiments shall be conducted for 50 marks and scaled down to **10 marks.**
5. Scaled-down marks of write-up evaluations and tests added will be CIE marks for the laboratory component of IPCC for **25 marks**.
6. The student has to secure 40% of 25 marks to qualify in the CIE of the practical component of the IPCC.

SEE for IPCC

Theory SEE will be conducted by University as per the scheduled timetable, with common question papers for the course (**duration 03 hours**)

1. The question paper will have ten questions. Each question is set for 20 marks.
2. There will be 2 questions from each module. Each of the two questions under a module (with a maximum of 3 sub-questions), **should have a mix of topics** under that module.
3. The students have to answer 5 full questions, selecting one full question from each module.
4. Marks scored by the student shall be proportionally scaled down to 50 Marks.

The theory portion of the IPCC shall be for both CIE and SEE, whereas the practical portion will have a CIE component only. Questions mentioned in the SEE paper may include questions from the practical component Semester End Exam (SEE): 100 marks (students have to answer all main questions) which will be reduced to 50 Marks.

CO to PO Mapping

PO1: Science and engineering Knowledge	**PO7:** Environment and Society
PO2: Problem Analysis	**PO8:** Ethics
PO3: Design & Development	**PO9:** Individual & Team Work
PO4: Investigations of Complex Problems	**PO10:** Communication
PO5: Modern Tool Usage	**PO11:** Project Mgmt. & Finance
PO6: Engineer & Society	**PO12:** Lifelong Learning

PSO1: Understand fundamental and advanced concepts in the core areas of Computer Science and Engineering to analyze, design and implement the solutions for the real world problems.

PSO2: Utilize modern technological innovations efficiently in various applications to work towards the betterment of society and solve engineering problems

LO: At the end of this session the student will be able to 1. Understands the basics of Memory Hierarchy and Cache Memory. 2. Understands theCACHE Architecture. 3. Understand CACHE policy. 4. Understand The Relationship between Cache and Main Memory.	PO3-2 PO5-3 PO6-2 PO12-2 PSO1-3 PSO2-2

Text Books

1. Andrew N Sloss, Dominic Symes and Chris Wright, ARM system developers guide, Elsevier, Morgan Kaufman publishers, 2008.

Reference Books

1. Raghunandan.G.H, Microcontroller (ARM) and Embedded System, Cengage learning Publication, 2019.
2. Insider's Guide to the ARM7 based microcontrollers, Hitex Ltd.,1st edition, 2005

Additional Reading

David E. Culler, Jaswinder Pal Singh, Anoop Gupta: Parallel Computer Architecture, A Hardware / Software Approach, Morgan Kaufman, 1999

Useful Websites

https://nptel.ac.in/courses/117/104/117104072/
https://nptel.ac.in/courses/117/106/117106112/
https://www.coursera.org

Useful Journals

- American Journal of Embedded System and Applications.
- Journal of Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)

Teaching and Learning Methods

1. Lecture class: 40 hrs
2. Practical: 20 hrs

Assessment Details (both CIE and SEE)

The weightage of Continuous Internal Evaluation (CIE) is 50% and for Semester End Exam (SEE) is 50%. The minimum passing mark for the CIE is 40% of the maximum marks (20 marks out of 50) and for the SEE minimum passing mark is 35% of the maximum marks (18 out of 50 marks). A student shall be deemed to have satisfied the academic requirements and earned the credits allotted to each subject/ course if the student secures a minimum of 40% (40 marks out of 100) in the sum total of the CIE (Continuous Internal Evaluation) and SEE (Semester End Examination) taken together.

CIE for the theory component of the IPCC (maximum marks 50)

- IPCC means practical portion integrated with the theory of the course.
- CIE marks for the theory component are **25 marks** and that for the practical component is **25 marks**.
- 25 marks for the theory component are split into **15 marks** for two Internal Assessment Tests (Two Tests, each of 15 Marks with 01-hour duration, are to be conducted) and **10 marks** for other assessment methods mentioned in 22OB4.2. The first test at the end of 40-50% coverage of the syllabus and the second test after covering 85-90% of the syllabus.
- Scaled-down marks of the sum of two tests and other assessment methods will be CIE marks for the theory component of IPCC (that is for **25 marks**).

CO	PO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2
BCS402	K-level														
CO1	K3	3	2	2	--	3	2	--	--	--	--	--	2	3	2
CO2	K3	3	2	3	--	3	2	--	--	--	--	--	2	3	2
CO3	K3	3	3	2	--	3	2	--	--	--	--	--	2	3	2
CO4	K3	3	3	3	--	3	2	--	--	--	--	--	2	3	2
CO5	K3	3	3	2	--	3	2	--	--	--	--	--	2	3	2

Course In charge

HOD-CSE

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

IQAC Coordinator

Principal

Dr. K. RAMA NARASIMHA
Principal/Director
K S School of Engineering and Managem
Bengaluru - 560 109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

FIRST ASSIGNMENT

Degree	:	**B.E**	**Semester**	:	**IV**
Branch	:	**CSE**	**Course Code**	:	**BCS402**
Course Title	:	**Microcontrollers**	**Max Marks**	:	**25**
Date	:	**20/05/2024**	**Last Date for submission**	:	**25/05/2024**

Q No.	Questions	Marks	K-Level	CO mapping
1	**a) Demonstrate** the embedded system hardware and software with a neat block diagram. **b) Illustrate** pipeline process of ARM processor with an example. **c) Interpret** Exception, interrupt and vector table.	5	Applying K3	CO1
2	**a)** Draw and **Explain** data flow diagram (architectural diagram) of ARM. **b) Explain** CPSR in detail with neat diagram.	5	Understanding K2	CO1
	a) Summarize complete ARM register set. **b) Differentiate** the following i. Microprocessor and Microcontroller ii. Harvard and Von-Neumann architecture with diagram. iii. CISC and RISC processors. **c) Discuss** the core extensions for ARM processor	5	Understanding K2	CO1
4	**a) Design** an assembly program to find the sum of first 10 integer numbers. **b)** With a neat diagram and shift instructions, **illustrate** the working of Barrel shifter with example.	5	Applying K3	CO2
5	**a) Determine** the working of data Processing instructions of ARM with syntax and examples. **b) Demonstrate** the working of branch instructions with syntax and example.	5	Applying K3	CO2

Course In charge

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

SESSION: 2023-2024 (EVEN SEMESTER)

SECOND ASSIGNMENT

Degree	:	**B.E**	**Semester**	:	**IV**
Branch	:	**CSE**	**Course Code**	:	**BCS402**
Course Title	:	**Microcontrollers**	**Max Marks**	:	**25**
Date	:	**26/06/2024**	**Last Date for submission**	:	**03/07/2024**

Q No.	Questions	Marks	K-Level	CO mapping

1	**a) Identify** the working of Load and Store instructions with examples.
	b) Make use of software interrupt instruction to solve the given problem
	`PRE cpsr = nzcVqifT_USER`
	`pc = 0x00008000`
	`lr = 0x003fffff ; lr = r14`
	`r0 = 0x12`
	`0x00008000 SWI 0x45`
		5	Applying K3	CO2

2	**a) Determine** the operation of program status register instructions with an example. **b) Illustrate** coprocessor instructions with syntax and example. **c) Interpret** loading constants with syntax.	5	Applying K3	CO2
3	**a) Write a note** on register allocation. **b) Write a note** function call. **c) Discuss** various Portability issues in detail.	5	Understanding K2	CO3
4	**a) Illustrate** the importance of basic C data types with an example. **b) Interpret** C looping structure with an example.	5	Applying K3	CO3
5	**a) Illustrate** pointer aliasing with an example. **b) Interpret** structure arrangement with an example.	5	Applying K3	CO3

26/6/24

Course In charge

HOD

HOD
Department of Computer Science Engineering
K.S.School of Engineering & Management

Model Question Paper-1/2 with effect from 2023-24

USN | | | | | | | | | | |

Fourth Semester B.E. Degree Examination
Subject Title: Microcontrollers

TIME: 03 Hours **Max. Marks: 100**

Note: 01. Answer any **FIVE** full questions, choosing at least **ONE** question from each **MODULE**.

		Module -1	*Bloom's Taxonomy Level	Marks
Q.01	a	Mention the difference between 1. Microctroller and Microprocessor 2. RISC and CISC	L2	10 Marks
	b	Explain the architecture of an ARM embedded device with the help of neat diagram.	L2	10 Marks
		OR		
Q.02	a	Explain in detail about Current Program Status Register (CPSR).	L2	10 Marks
	b	With a neat diagram, explain embedded system Hardware.	L2	10 Marks
		Module-2		
Q. 03	a	Explain different arithmetic instructions in ARM processor with an example.	L2	10 Marks
	b	Explain single register load store addressing mode syntax, table, index mode with an example.	L2	10 Marks
		OR		
Q.04	a	Explain barrel shifter instructions in ARM with suitable examples.	L2	10 Marks
	b	Explain different Logical instructions in ARM processor with an example.	L2	10 Marks
		Module-3		
Q. 05	a	Explain code optimization, profiling and cycle counting.	L3	10 Marks
	b	Write a C program that prints the square of the integers between 0 to 9 using functions and explain how to convert this C function to an assembly function with command.	L3	10 Marks
		OR		
Q. 06	a	Discuss how registers are allocated to optimize the program.	L3	10 Marks
	b	Develop an ALP to find the sum of first 10 integer numbers.	L3	10 Marks
		Module-4		
Q. 07	a	With a neat diagram explain ARM processor exceptions and modes.	L2	10 Marks
	b	Explain assigning interrupts and interrupt latency.	L2	10 Marks
		OR		
Q. 08	a	Briefly explain what happens when an IRQ and FIQ exception is raised with an ARM processor.	L2	10 Marks

	b	Explain firmware execution flow and explain Red Hat RedBoot.	L2	10 Marks
		Module-5		
Q. 09	a	Explain the basic architecture of cache memory.	L2	10 Marks
	b	Explain how main memory maps to a cache memory.	L2	10 Marks
		OR		
Q. 10	a	With a neat block diagram explain associative cache.	L2	10 Marks
	b	Briefly explain cache line replacement policies.	L2	10 Marks

*Bloom's Taxonomy Level: Indicate as L1, L2, L3, L4, etc. It is also desirable to indicate the COs and POs to be attained by every bit of questions.

CBCS SCHEME

USN | 1 | K | G | 2 | 2 | C | S | 1 | 0 | G |

BCS402

Fourth Semester B.E./B.Tech. Degree Examination, June/July 2024
Microcontrollers

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

			M	L	C
		Module – 1			
Q.1	a.	Explain the architecture of an arm embedded device with a neat diagram.	10	L2	CO1
	b.	How are monitor and control internal operations performed in ARM core? Explain in brief.	10	L2	CO1
		OR			
Q.2	a.	Explain memory management in ARM core. Compare cache and tightly coupled memory.	10	L2	CO1
	b.	Explain mechanism applied by ARM core to handle exception, interrupts using different vector table.	10	L2	CO1
		Module – 2			
Q.3	a.	Examine data processing instructions requirement in the manipulation of data register? Explain in brief data processing instructions.	10	L2	CO2
	b.	Explain with examples the following 32-bit instruction of ARN processor i) CMN ii) MLA iii) MRS iv) BIC v) LDR.	10	L2	CO2
		OR			
Q.4	a.	Explain the following with example : i) Stock operation ii) Swap instructions.	10	L2	CO2
	b.	Expiain Branch instructions in ARM with suitable example. Demonstrate Branch instruct usage flow of execution with an example program.	10	L2	CO2
		Module – 3			
Q.5	a.	How registers are allocated to optimize the program? Develop an assembly level program to find the sum of first to integer numbers.	10	L2	CO3
	b.	How complier handles a "for loop" with variable number of iterations N and loop controlling with an example.	10	L2	CO3
		OR			
Q.6	a.	Explain the following terms with an appropriate example : i) Pointer Aliasing ii) Portability issues.	10	L2	CO3
	b.	How function calling is efficiently used by ARM through APCS with an example program.	10	L2	CO3
		Module – 4			
Q.7	a.	Explain ARM processors exception and modes with a neat diagram.	10	L2	CO4
	b.	Explain exception priorities and link register offset.	10	L2	CO4
		OR			
Q.8	a.	List ARM firmware suite features. Explain firmware execution flow and Red Hat Boot.	10	L2	CO4
	b.	Explain IRQ and Fir exception, also to enable and disable IRQ and FIQ interrupts.	10	L2	CO4
		Module – 5			
Q.9	a.	Explain basic architecture of cache memory.	10	L2	CO5
	b.	Explain process involved in main memory mapping to a cache memory.	10	L2	CO5
		OR			
Q.10	a.	Explain with diagram set associative cache. How are efficiency is measured?	10	L2	CO5
	b.	Briefly explain cache line replacement policies with an example.	10	L2	CO5

* * * * *

Microcontrollers-BCS402

Module - 1(Chapter-7): ARM PROCESSOR BASICS

Evolution of ARM Processors:

- 1985, First ARM (ARM1)
- 1995, ARM7TDMI
  - Most successful ARM core
  - 3-stage pipeline, 120 Dhrystone MIPS
- 1997, ARM9
  - 5-stage pipeline
  - Harvard (I+D cache), MMU (OS's VM)
- 1999, ARM10
  - 6-stage pipeline
  - VFP (Vector Float Point) (7-stage pipeline)
- 2003, ARM11
  - 8-stage pipeline

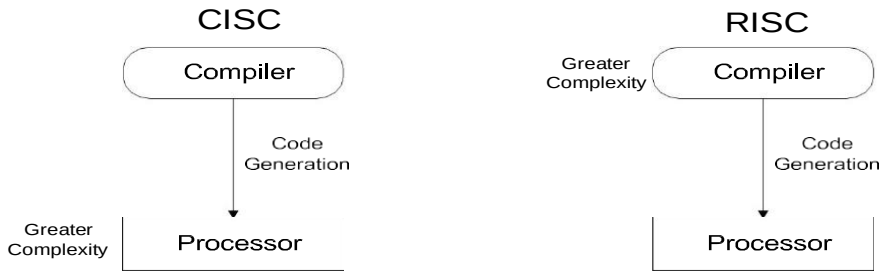
Versions of ARM Architecture

- ARMv1
  - 26-bit address
- ARMv2
  - 32-bit Multiplier/coprocessor
- ARMv3
  - 32-bit address, cpsr/spsr, MMU, undef/abort Mode
- ARMv4
  - Load/store (sign/half/byte), sys Mode
- ARMv5
  - Superset ARMv4T (Thumb), extend Mul/DSP
- ARMv6
  - Multiprocessor support instr., unaligned/endian/MMX
  - Others
- StrongARM
  - ARM + Digital Semiconductor
  - Intel Patent
- Xscale
  - 1GHz, V5TE
- SC100
  - Security, Low Power
  - ARM7TDMI, MPU

Philosophy of RISC design

RISC is a design philosophy aimed at delivering simple but powerful instructions that execute within a single cycle at a high clock speed. The RISC philosophy concentrates on reducing the complexity of instructions performed by the hardware. The RISC philosophy provides greater flexibility and intelligence in software rather than

hardware. RISC design places greater functionality to the compiler rather than the hardware.
The RISC philosophy is implemented with four major design rules:



- Instruction
  - RISC processor have a Reduced number of instruction classes. These classes provide simple operations that can each execute in a single cycle. The compiler or programmer synthesizes complicated operations (e.g. a divide operation) by combining several simple instructions. Each instruction is of the same length to support pipelining. But CISC instructions are of variable length.
- Pipeline
  - The processing of instructions is broken down into smaller units (stage) that can be executed in parallel by pipelines. There is no need for an instruction to be executed by a mini-program (microcode) as on CISC processor.
- Register
  - RISC have a large General Purpose Registers (GPR) set. Any of these registers can hold either data or an address. But CISC processors have dedicated registers for specific purposes.
- Load/store architecture
  - Only Load and Store instructions are used to transfer data between the register bank and external memory.
  - It separates memory access from data processing and allows multiple use of the data items held in the register bank without accessing memory each time. But in CISC design data processing can operate on the memory directly.

The ARM Design Philosophy

- There are a number of physical features that have driven the ARM processor design:
  - Low Power Consumption: Smallest Core;
  - Limited Memory: High code density;
  - Reduced area of the processor Die: Simple Hardware Executive Unit
  - Low cost memory devices
  - Built in H/w for debug technology
  - Total effective system performance.
- These design rules allow a RISC processor to be simpler, and thus the core can operate at higher clock frequencies.

- The ARM instruction set differs from the pure RISC definition in several ways
  - make the ARM suitable for embedded application
    - » Variable cycle execution for certain instruction
      - Not every ARM instruction executes in a single cycle. Load/store depends on no.of registers being transferred.
    - » Inline barrel shifter- leads to more complex instruction

- It is a hardware component that preprocesses one of the i/p registers before it is used by the instruction. This expands the capability of many instructions to improve the core performance and code density.

» Thumb 16-bit instruction set
- It is a second 16-bit instruction set that permits the ARM core to execute either 16 or 32-bit instructions
- The Thumb instruction improve code density by about 30%.

» Conditional execution
- Improves performance and code density by reducing Branch.

» Enhanced instruction
- DSP instruction were added to the standard ARM instruction-set to support fast 16x16-bit multiplier operations and saturation.

Embedded System Hardware

Embedded systems can control many different devices like small sensors, real-time control systems. Embedded systems are a combination of software and hardware components. Each component can be chosen or designed.

- An Embedded system device can be separated into four main components:
  - ARM Processor: controls the embedded device.
    - » An ARM processor comprises a core (the execution engine that processes instructions and manipulates data), plus the surrounding components (MMU and caches) that interface it with a bus.
  - Controllers: coordinate important functional blocks (e.g. interrupt and memory controllers)
  - Peripherals: Provide input output capability external to the chip. Peripherals are unique for each type of embedded device.
  - Bus: is used to communicate between different parts of the device.

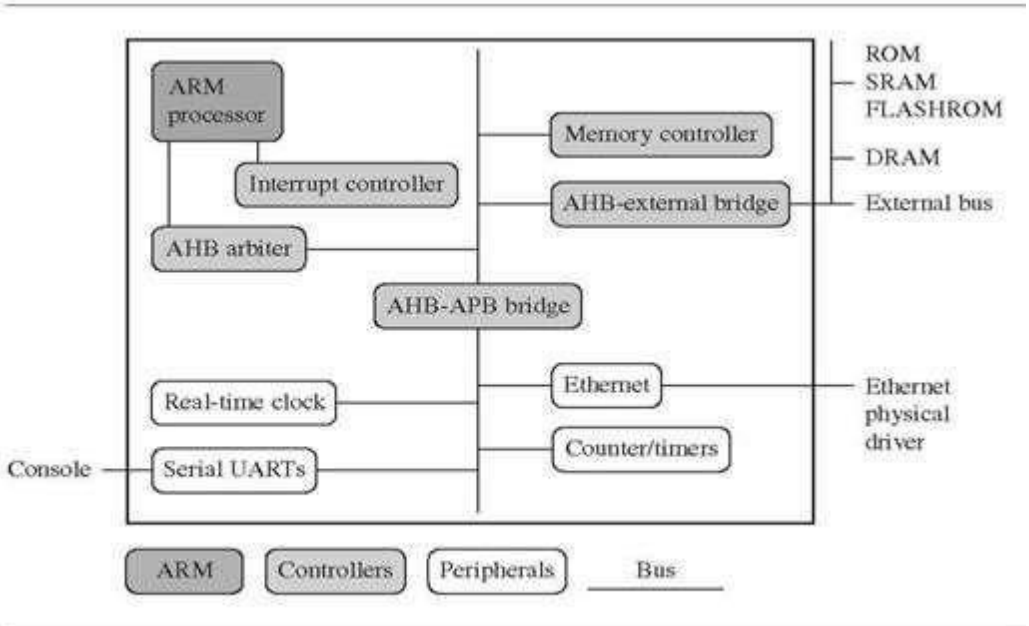


Figure of An example of an ARM-based embedded device, a microcontroller.

ARM Bus technology
- Embedded systems use different bus technologies than those designed for x86 PC.
  - X86 uses PCI bus technology connects Video cards and HD controllers and hence known as external or off-chip
  - Embedded device use an on-chip bus which is internal to the chip

- A Bus has two architecture levels
  - The First is a physical level that covers the *electrical characteristics* and *bus width* (16, 32, or 64 bits).
  - The Second level is the *protocol*– the logical rules governing the communication between processor and peripheral.
- ARM seldom implements the electrical characteristics of the bus, but it routinely specifies the bus protocol.

AMBA Bus Protocol

- AMBA Advanced Micro controller Bus Architecture
  - Introduced in 1996, it's widely adopted as the on-chip bus architecture for ARM processors.
  - The first AMBA buses introduced were
    - » ASB : ARM System Bus, and
    - » APB : ARM Peripheral Bus
  - Later, ARM introduced another bus design
    - » AHB: ARM High-performance Bus
- Using AMBA,
  - peripheral designers can reuse the same design on multiple projects (with different processor architecture).
  - Plug-and-play interface improves availability and time to market for hardware developers.
  -
- AHB
  - provides higher data throughput than ASB. Because
    - » It uses a Centralized Multiplexed Bus Scheme (rather than ASB's bi-direction bus).
    - » This change allows the AHB bus to run at higher clock speed.
    - » 64/128 bits width.
    - » Two variations on the AHB bus
    - » Multi-layer AHB, and
      - allows multiple active bus masters,
    - » AHB-Lite: only one master

Memory

- Memory is necessary to have some form of memory to store and execute code.
- For good memory characteristics compare : price, performance, and power consumption
- Specific memory characteristics are hierarchy, width, and type
- To double the speed for a required bandwidth, memory needs more power.

Memory hierarchy

- Cache
  - is used to speed up data transfer between Core and Main Memory (DRAM)
  - is physically located nearby the ARM processor core and it is the fastest memory
  - It provides an overall increase in performance, but does not support real time system response
    - » Note that many small embedded systems do not require the benefit of a cache.

- Main Memory
  - Is large and it is placed after Cache memory as it is slower than cache
  - Load store instructions access the main memory if the values are not in cache.
- Secondary storage

- It is the largest and slowest memory and is placed away from the main memory. Width adaptive (e.g., 32-bit Core vs. 16-bit BUS)

Width

- The memory width is the number of bits the memory returns on each access
- Typically they are 8,16,32,or 64 bits.
- Memory width directly effects the overall performance and cost ratio

Memory types

- DRAM
  - the most commonly used RAM for devices;
  - Dynamic: need to have its storage cells refreshed and given a new electronic charge every few milliseconds, so you need to set up a DRAM controlr before using the memory.
- SRAM
  - is faster than the more traditional DRAM (SRAM does not require a pause between data access).
- SDRAM
  - is one of many subcategories of DRAM.
  - accessed pipelined, transferred in a burst.

Peripherals

- Embedded system that interact with the outside world need some form of peripheral device.
  - Peripherals range from a simple serial communication device to a more complex 802.11 wireless device.
- All ARM peripherals are memory mapped – the programming interface is a set of memory addressed register.
- Controllers are specialized peripherals that implement higher level of functionality within an embedded system.
  - Two important types of controllers are
    - Memory Controller
    - Interrupt Controller
      - Normal IC
      - Vectoring IC
        - Priority
        - Simple Interrupt Dispatch

Memory Controllers: Connect different types of memory to the processor bus.

- On power-up a memory controller is configured in hardware to allow certain memory device to be active. These memory devices allow the initialization code to be executed.
- Some memory devices must be set up by software.
  - e.g. When using DRAM, you first have to set up the memory timings and refresh rate before it can be accessed.

Interrupt controller: When a peripheral or device requires attention,

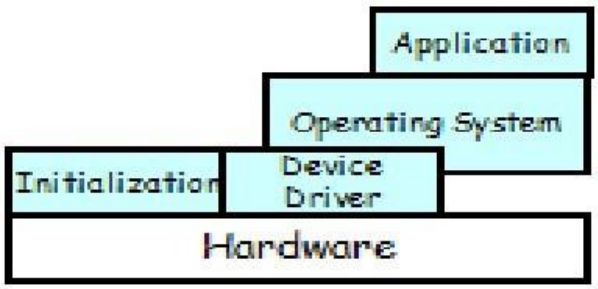
- it raise an interrupt to the processor.

- An interrupt controller
  - provides a programmable governing policy
  - There are two types of interrupt controller available for the ARM processor
  - Standard interrupt controller
    - Sends an interrupt signal; Can be programmed to ignore or mask an individual or set of devices.
    - It's interrupt handler determines which device requiring service.
  - Vector interrupt controller (VIC)
    - Associate a "priority" and a "handler address" to each interrupt.

- Depending on its type, VIC will either call the standard interrupt exception handler (loading the handler address from VIC) or cause coreto jump to the handler for the device directly.

Embedded System Software

- An embedded system needs software to drive it.
- There are four typical software components required to control an embedded device.
  - » Each software component in the stack uses a higher level of abstraction to separate the code from the hardware device.
  - Initialization Code (e.g. Boot loader)
  - Operating System
  - Device Drivers
  - Application



Initialization code (or boot code): is the first code executed on the board and is specific to a particular target or group of targets. It sets-up the minimum parts of the board before handing over the control to the operating system.

- takes the processor from the reset state to a state where the operating system can run.
  - » Configuring memory controller, caches
  - » Initializing some devices
  - » in a simple system the OS is replaced by a Debug Monitor or a simple scheduler.

- Three phases of tasks before handing over the control to the operating system are:
  - Initial hardware configuration
    - » Satisfy the requirements of the booted image
      - e.g. re-organization of the memory map
  - Diagnostics
    - » Fault identification and isolation
  - Booting
    - » Loading an image and handing control over to the image
    - » The boot process may be complicated if the system must boot different operating systems or different versions of the same operating system.

Example: Memory Reorganization

- Start from ROM
- Remap to RAM
  - easy IVT modification

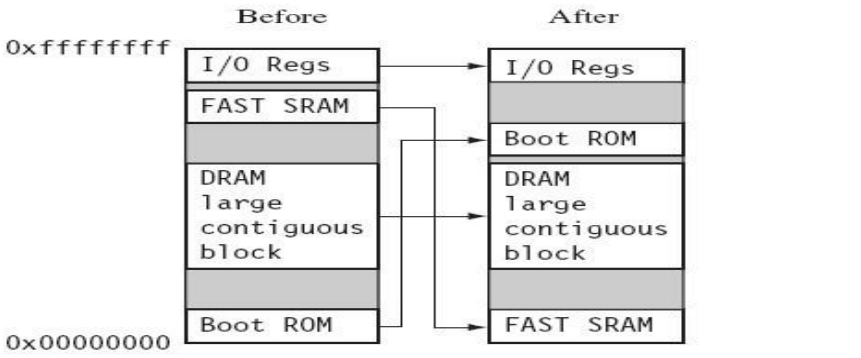


Figure of Memory remapping.

Operating Systems

- OS organizes the system resources
  - peripherals, memory, and processing time
    - » With an OS controlling these resources, they can be efficiently used by different applications running within the OS environment.
- ARM processors support over 50 OSes
  - Two main categories: RTOS, platform OS
    - » RTOS: guarantee response times to event
    - » platform OS: require MMU and tend to have secondary storage (for large application).
      - N.B., These two categories of OSes are not mutually exclusive.
  - ARM has developed a set of processor cores that specially target each category.

Applications:

- The OS schedules applications
  - code dedicated to handling a particular task.
- ARM processors are found in numerous market segments, including
  - networking, automotive, mobile and consumer devices, mass storage, and imaging.
- In contrast, ARM processors are not found in applications that require leading-edge high performance.

ARM core dataflow model

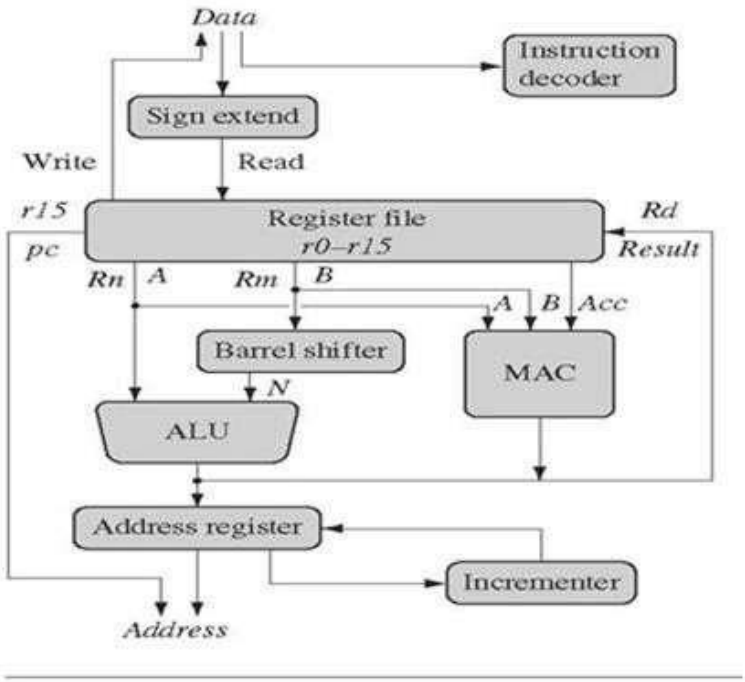


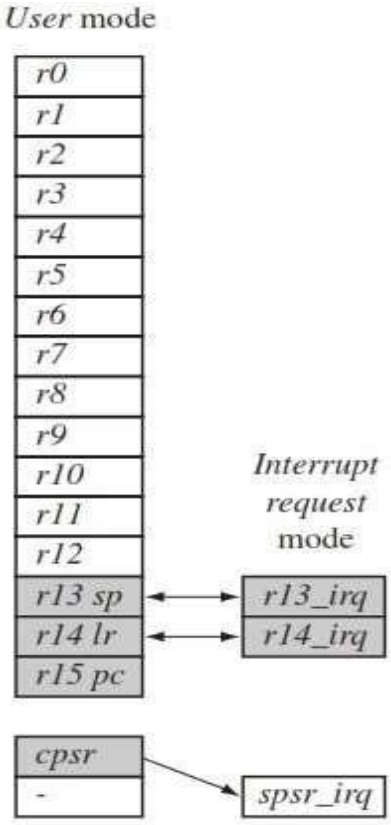
Figure 7.1 ARM core dataflow model.

- ARM core as functional units connected by data buses,
- Arrows represent the flow of data.
- lines represent the buses.
- Boxes represent either an operation unit or a storage area
- Data enters the processor core through the *Data* bus
- data may be an instruction to execute or a data item.
- Von Neumann implementation of the ARM— data items and instructions share the same bus.
- Harvard implementations of the ARM use two different buses.
- Instruction decoder translates instructions before they are executed
- ARM processor, like all RISC processors, uses a *load-store architecture*.
- This means it has two instruction types for transferring data in and out of the processor
- load instructions copy data from memory to registers in the core
- store data from registers to memory.
- There are no data processing instructions that directly manipulate data in memory.
- Data items are placed in the *register file*—a storage bank made up of 32-bit registers
- The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
- ARM instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*.
- Source operands are read from the register file using the internal buses *A* and *B*, respectively.
- ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result.
- Data processing instructions write the result in *Rd* directly to the register file.
- Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.
- load instructions copy data from memory to registers in the core
- store data from registers to memory.
- There are no data processing instructions that directly manipulate data in memory.

- Data items are placed in the *register file*—a storage bank made up of 32-bit registers
- The sign extend hardware converts signed 8-bit and 16-bit numbers to 32-bit values as they are read from memory and placed in a register.
- ARM instructions typically have two source registers, *Rn* and *Rm*, and a single result or destination register, *Rd*.
- Source operands are read from the register file using the internal buses *A* and *B*, respectively.
- ALU (arithmetic logic unit) or MAC (multiply-accumulate unit) takes the register values *Rn* and *Rm* from the *A* and *B* buses and computes a result.
- Data processing instructions write the result in *Rd* directly to the register file.
- Load and store instructions use the ALU to generate an address to be held in the address register and broadcast on the *Address* bus.
- An important feature of ARM is the Barrel Shifter. The register *Rm* can be pre-processed in this Barrel shifter, before it enters ALU. Thus Barrel shifter and ALU together can calculate a wide range of expressions and addresses.
- After processing the result in Rd is written back to the register file using the result bus.
- For load-store instructions the incrementer updates the address register before the core reads or writes the next register value from or to the next sequential memory location.
- The processor keeps executing until an exception or interrupt changes the normal execution flow.

REGISTERS

- General-purpose registers hold either data or an address.
- They are identified with the letter *r* prefixed to the register number.
- Figure below shows the active registers available in *user* mode—a protected mode normally
- The processor can operate in seven different modes
- 18 active registers: 16 data registers and 2 processor status registers.
- data registers - *r0* to *r15*
- Three registers assigned to a particular task or special function: *r13*, *r14*, and *r15*
- Register *r13* is traditionally used as the stack pointer (*sp*) and stores the head of the stack in the current processor mode.
- Register *r14* is called the link register (*lr*) and is where the core puts the return address whenever it calls a subroutine.



- Register *r15* is the program counter (*pc*) and contains the address of the next instruction to be fetched by the processor.
- registers *r13* and *r14* can also be used as general-purpose registers
- OS assumes r13 is pointing to valid stack frame, not recommended as general purpose registers

Register in ARM:

- Orthogonal Registers (ref. VAX, PDP-11)
  - We say R0~R13 are orthogonal, for given instruction, if it can use R0, then others can also be used.
  - there are two program status registers: *cpsr* and *spsr* (the current and saved program status registers, respectively).

PSRs

  - R13(sp), R14(lr), R15(pc)
- CPSR/SPSR
  - Condition Codes: N, Z, C, V
  - Interruption mask: I(IRQ), F(FIQ)
  - Thumb Enable Bit
  - Mode(5-bit)

Current Program Status Register:

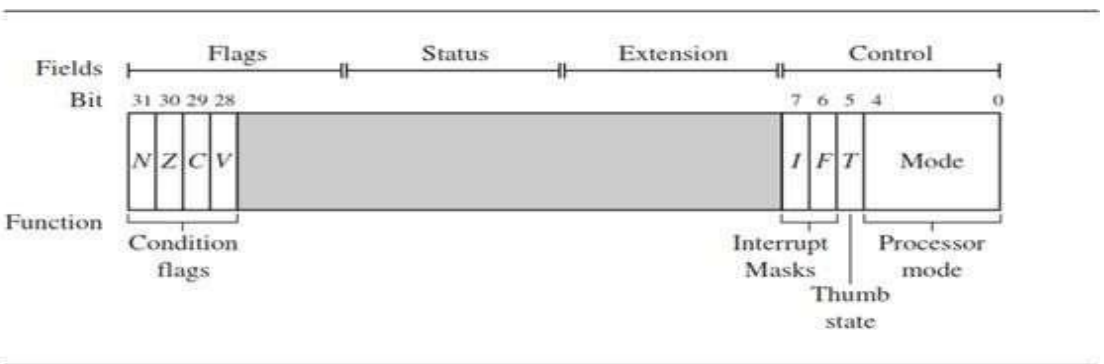


Figure 7.3 A generic program status register (*psr*).

- ARM core uses the cpsr to monitor and control internal operations.
- cpsr is a dedicated 32-bit register and resides in the register file.
- Figure shows the basic layout of a generic program status register.
- Note that the shaded parts are reserved for future expansion.
- The cpsr is divided into four fields, each 8 bits wide: flags, status, extension, and control
- In current designs the extension and status fields are reserved for future use .
- The control field contains the processor mode, state, and interrupt mask bits.
- The flags field contains the condition flags.
- Some ARM processor cores have extra bits allocated.
- For example, the J bit, which can be found in the flags field, is only available on Jazelle-enabled processors, which execute 8-bit instructions

Processor Modes

- Processor mode determines which registers are active and the access rights to the cpsr register itself.
- Each processor mode is either privileged or nonprivileged.
- A privileged mode allows full read-write access to the cpsr.
- Conversely, a nonprivileged mode only allows read access to the control field in the cpsr but still allows read-write access to the condition flags
- There are seven processor modes in total: six privileged modes (abort, fast interrupt request, interrupt request, supervisor, system, and undefined) and one nonprivileged mode(user),
- Processor enters **abort mode** when there is a **failed attempt to access memory**.
- **Fast interrupt request and interrupt request modes correspond** to the **two interrupt levels** available on the ARM processor.
- **Supervisor mode** is the mode that the processor is in **after reset** and is generally the mode that an **operating system kernel operates in**.
- **System mode** is a **special version of user mode that allows full read-write access** to the cpsr.
- **Undefined mode** is used **when the processor encounters an instruction that is undefined or not supported** by the implementation.
- **User mode** is used **for programs and applications**.

Banked Registers

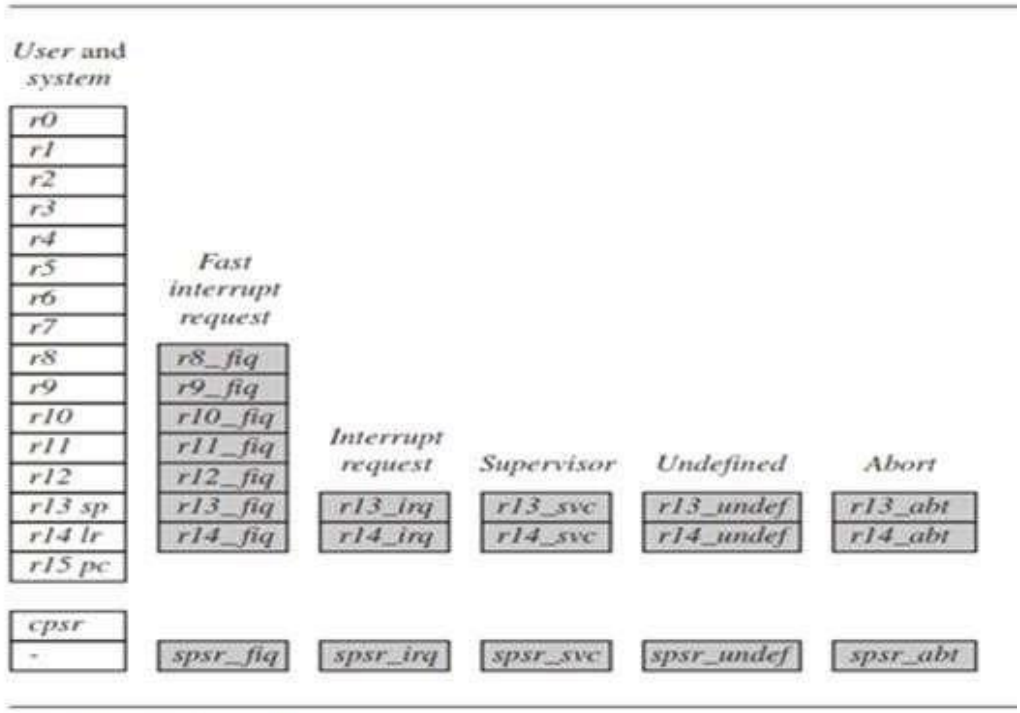


Figure 7.4 Complete ARM register set.

- Figure shows all 37 registers in the register file.
- 20 registers are hidden from a program at different times.
- These registers are called *banked registers* and are identified by the shading in the diagram.
- They are available only when the processor is in a particular mode.
- For example, *abort* mode has banked registers *r13_abt*, *r14_abt* and *spsr_abt*.
- Banked registers of a particular mode are denoted by an underline character post-fixed to the mode mnemonic or *_mode*.
- Every processor mode except *user* mode can change mode by writing directly to the mode bits of the *cpsr.*
- All processor modes except *system* mode have a set of associated banked registers that are a subset of the main 16 registers.
- A banked register maps one-to-one onto a *user* mode register.
- If you change processor mode, a banked register from the new mode will replace an existing register.
- For example, when the processor is in the *interrupt request* mode, the instructions you execute still access registers named *r13* and *r14*.

- However, these registers are the banked registers *r13_irq* and *r14_irq*.
- The *user* mode registers *r13_usr* and *r14_usr* are not affected by the instruction referencing these registers.
- A program still has normal access to the other registers *r0* to *r12*.
- processor mode can be changed by a program that writes directly to the *cpsr* (the processor core has to be in privileged mode) or by hardware when the core responds to an exception or interrupt.
- The **following exceptions and interrupts cause a mode change: *reset*, *interrupt request*, *fast interrupt request*, *software interrupt*, *data abort*, *prefetch abort*, and *undefined instruction*.**
- Exceptions and interrupts suspend the normal execution of sequential instructions and jump to a specific location

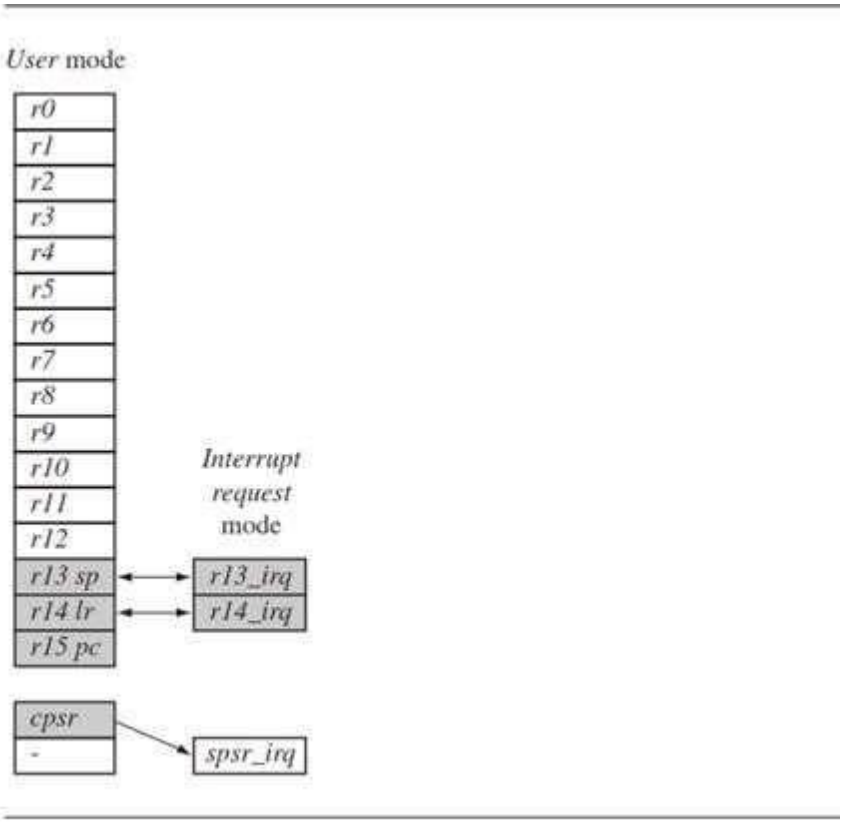


Figure 7.5 Changing mode on an exception.

- The figure shows the core changing from *user* mode to *interrupt request* mode.
- This change causes *user* registers *r13* and *r14* to be banked.
- The *user* registers are replaced with registers *r13_irq* and *r14_irq*, respectively.
- Note *r14_irq* contains the return address and *r13_irq* contains the stack pointer for *interrupt request* mode.
- Figure also shows a new register appearing in *interrupt request* mode
- The **saved program status register** *(spsr)*, which **stores the previous mode's *cpsr***.
- Can see in the diagram the *cpsr* being copied into *spsr_irq*.
- To return back to *user* mode, a special return instruction is used that instructs the core to restore the original *cpsr* from the *spsr_irq* and bank in the *user* registers *r13* and *r14*.
- Note that **the *spsr* can only be modified and read in a privileged mode**. There is no *spsr* available in *user* mode.
- Another important feature to note is that the *cpsr* is not copied into the *spsr* when a mode change is forced due to a program writing directly to the *cpsr*.
- The **saving of the *cpsr* occurs only when an exception or interrupt is raised**.
- The table below shows that the current active processor mode occupies the five least significant bits of the *cpsr*.
- When power is applied to the core, it starts in *supervisor* mode.

List of various modes and the associated binary patterns.

Table 7.1 Processor mode.

Mode	Abbreviation	Privileged	Mode[4:0]
Abort	abt	yes	10111
Fast interrupt request	fiq	yes	10001
Interrupt request	irq	yes	10010
Supervisor	svc	yes	10011
System	sys	yes	11111
Undefined	und	yes	11011
User	usr	no	10000

The last column of the table gives the bit patterns that represent each of the processor modes in the *cpsr*.

State and Instruction Sets

Table 7.2 ARM and Thumb instruction set features.

	ARM (*cpsr* $T = 0$)	Thumb (*cpsr* $T = 1$)
Instruction size	32-bit	16-bit
Core instructions	58	30
Conditional execution[a]	most	only branch instructions
Data processing instructions	access to barrel shifter and ALU	separate barrel shifter and ALU instructions
Program status register	read-write in privileged mode	no direct access
Register usage	15 general-purpose registers +*pc*	8 general-purpose registers +7 high registers +*pc*

- The state of the core determines which instruction set is being executed.
- There are three instruction sets: ARM, Thumb, and Jazelle.
- The ARM instruction set is only active when the processor is in ARM state.
- Thumb instruction set is only active when the processor is in Thumb state. Once in Thumb state the processor is executing purely Thumb 16-bit instructions.
- You cannot intermingle sequential ARM, Thumb, and Jazelle instructions.
- Jazelle *J* and Thumb *T* bits in the *cpsr* reflect the state of the processor.
- When both *J* and *T* bits are 0, the processor is in ARM state and executes ARM instructions.
- This is the case when power is applied to the processor.
- When the T bit is 1, then the processor is in Thumb state.
- To change states the core executes a specialized branch instruction. Table 2.2 compares the ARM and Thumb instruction set features.
- ARM designers introduced a third instruction set called *Jazelle.*
- *Jazelle* executes 8-bit instructions and is a hybrid mix of software and hardware designed to speed up the execution of Java bytecodes.
- To execute Java bytecodes, you require the Jazelle technology plus a specially modified version of the Java virtual machine.
- Note that the hardware portion of Jazelle only supports a subset of the Java bytecodes; the rest are emulated in software.

Table 7.3 Jazelle instruction set features.

	Jazelle (*cpsr* $T = 0, J = 1$)
Instruction size	8-bit
Core instructions	Over 60% of the Java bytecodes are implemented in hardware; the rest of the codes are implemented in software.

The Jazelle instruction set is a closed instruction set and is not openly available. Table 7.3 gives the Jazelle instruction set features.

Interrupt Masks

- Interrupt masks are used to stop specific interrupt requests from interrupting the processor.
- There are two interrupt request levels available on the ARM processor core—
  - *interrupt request* (IRQ)
  - *fast interrupt request* (FIQ).

- *cpsr* has two interrupt mask bits, 7 and 6 (or *I* and *F*), which control the masking of IRQ and FIQ.
- The *I* bit masks IRQ when set to binary 1, *F* bit masks FIQ when set to binary 1.

Condition Flags

Table 7.4 Condition flags.

Flag	Flag name	Set when
Q	Saturation	the result causes an overflow and/or saturation
V	oVerflow	the result causes a signed overflow
C	Carry	the result causes an unsigned carry
Z	Zero	the result is zero, frequently used to indicate equality
N	Negative	bit 31 of the result is a binary 1

- Condition flags are updated by comparisons and the result of ALU operations that specify the S instruction suffix.
- For example, if a SUBS subtract instruction results in a register value of zero, then the *Z* flag in the *cpsr* is set. This particular subtract instruction specifically updates the *cpsr.*
- With processor cores that include the DSP extensions, the *Q* bit indicates if an overflow or saturation occurs.
- The flag is "sticky" in the sense that the hardware only sets this flag. To clear the flag you need to write to the *cpsr* directly.
- In Jazelle-enabled processors, the *J* bit reflects the state of the core; if it is set, the core is in Jazelle state.
- The *J* bit is not generally usable and is only available on some processor cores.
- To take advantage of Jazelle, extra software has to be licensed from both ARM Limited and Sun Microsystems.
- Most ARM instructions can be executed conditionally on the value of the condition flags. Table 2.4 lists the condition flags and a short description on what causes them to be set.
- These flags are located in the most significant bits in the *cpsr*. These bits are used for conditional execution.

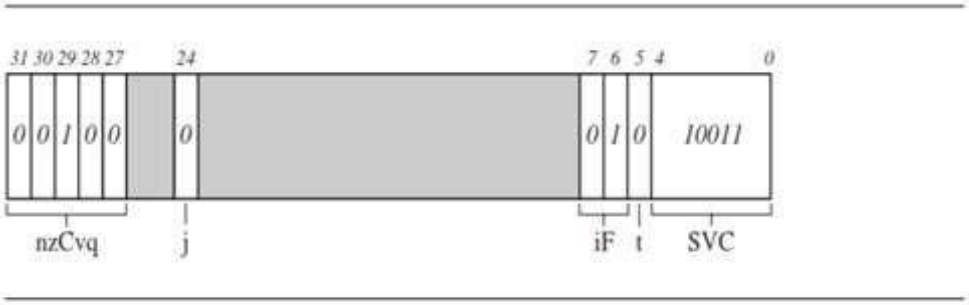


Figure 7.6 Example: *cpsr* = *nzCvqjiFt_SVC.*

- Figure 2.6 shows a typical value for the *cpsr* with both DSP extensions and Jazelle.
- When a bit is a binary 1 we use a capital letter.
- when a bit is a binary 0, we use a lowercase letter.
- For the condition flags a capital letter shows that the flag has been set.
- For interrupts a capital letter shows that an interrupt is disabled.
- In the *cpsr* the *C* flag is the only condition flag set.
- The rest *nzvq* flags are all clear.
- The processor is in ARM state because neither the Jazelle *j* nor Thumb *t* bits are set.
- IRQ interrupts are enabled, and FIQ interrupts are disabled.
- Finally can see from the figure the processor is in *supervisor* (*SVC*) mode since the mode[4:0] is equal to binary 10011.

Conditional Execution

Table 7.5 Condition mnemonics.

Mnemonic	Name	Condition flags
EQ	equal	*Z*
NE	not equal	*z*
CS HS	carry set/unsigned higher or same	*C*
CC LO	carry clear/unsigned lower	*c*
MI	minus/negative	*N*
PL	plus/positive or zero	*n*
VS	overflow	*V*
VC	no overflow	*v*
HI	unsigned higher	*zC*
LS	unsigned lower or same	*Z* or *c*
GE	signed greater than or equal	*NV* or *nv*
LT	signed less than	*Nv* or *nV*
GT	signed greater than	*NzV* or *nzv*
LE	signed less than or equal	*Z* or *Nv* or *nV*
AL	always (unconditional)	ignored

- Conditional execution controls whether or not the core will execute an instruction.
- Most instructions have a condition attribute.
- It determines if the core will execute it based on the setting of the condition flags.
- Prior to execution, the processor compares the condition attribute with the condition flags in the *cpsr*.
- If they match, then the instruction is executed; otherwise the instruction is ignored.

Pipeline

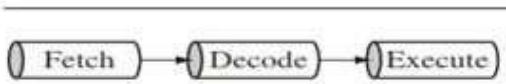


Figure 7.7 ARM7 Three-stage pipeline.

- A pipeline is the mechanism a RISC processor uses to execute instructions.
- Using a pipeline speeds up execution by fetching the next instruction while other instructions are being decoded and executed.

  Three-stage pipeline:
  - *Fetch* loads an instruction from memory.
  - *Decode* identifies the instruction to be executed.
  - *Execute* processes the instruction and writes the result back to a register.

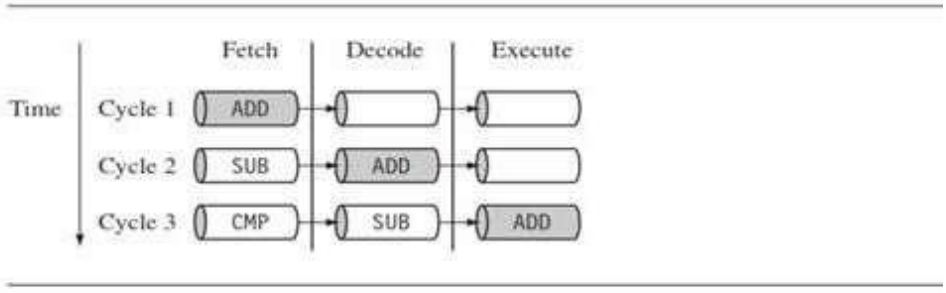


Figure 7.8 Pipelined instruction sequence.

- EX: shows a sequence of three instructions being fetched, decoded, and executed by the processor.
- Each instruction takes a single cycle to complete after the pipeline is filled.
- The three instructions are placed into the pipeline sequentially.
- In the first cycle the core fetches the ADD instruction from memory.
- In the second cycle the core fetches the SUB instruction and decodes the ADD instruction.
- In the third cycle, both the SUB and ADD instructions are moved along the pipeline.
- ADD instruction is executed.
- SUB instruction is decoded.
- CMP instruction is fetched
- This procedure is called *filling the pipeline*.
- The pipeline allows the core to execute an instruction every cycle.
- As pipeline length increases, the amount of work done at each stage is reduced, which allows the processor to attain a higher operating frequency.
- In turn increases the performance.
- System *latency* also increases because it takes more cycles to fill the pipeline before the core can execute an instruction.
- Increased pipeline length also means there can be data dependency between certain stages.

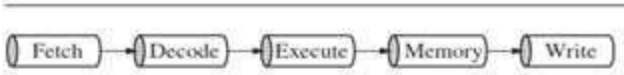


Figure 7.9 ARM9 five-stage pipeline.

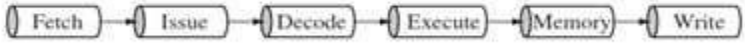


Figure 7.10 ARM10 six-stage pipeline.

- Pipeline design for each ARM family differs. For example, The ARM9 core increases the pipeline length to five stages.
- ARM9 adds a memory and write back stage, which allows the ARM9 to process on average 1.1 Dhrystone MIPS per MHz & increases throughput by around 13% compared with an ARM7.
- The maximum core frequency attainable using an ARM9 is also higher.
- ARM10 increases the pipeline length still further by adding a sixth stage
- Average 1.3 Dhrystone MIPS per MHz,
- 34% more throughput than an ARM7 processor core, but again at a higher latency cost.
- Even though the ARM9 and ARM10 pipelines are different, they still use the same *pipeline executing characteristics* as an ARM7.
- Code written for the ARM7 will execute on an ARM9 or ARM10

Pipeline Executing Characteristics

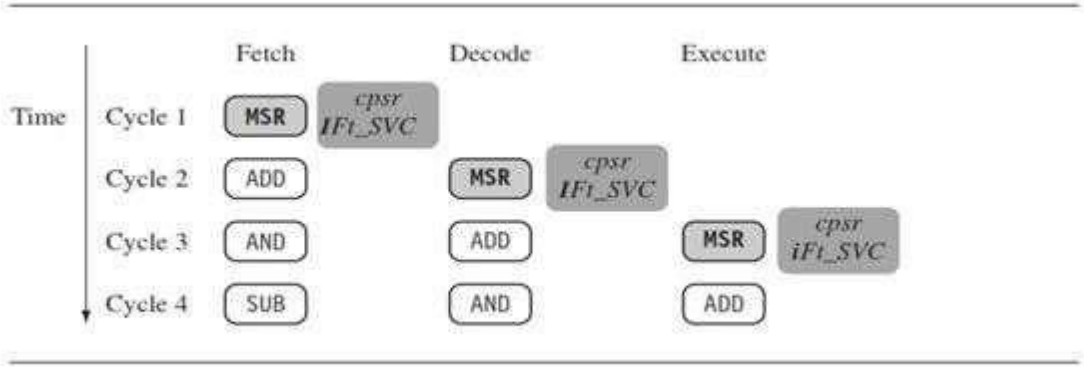


Figure 7.11 ARM instruction sequence.

- ARM pipeline has not processed an instruction until it passes completely through the execute stage.
- For example, an ARM7 pipeline (with three stages) has executed an instruction only when the fourth instruction is fetched.
- Figure 7.11 shows an instruction sequence on an ARM7 pipeline.
- MSR instruction is used to enable IRQ interrupts.
- Only occurs once the MSR instruction completes the execute stage of the pipeline.
- It clears the *I* bit in the *cpsr* to enable the IRQ interrupts.
- Once the ADD instruction enters the execute stage of the pipeline, IRQ interrupts are enabled.

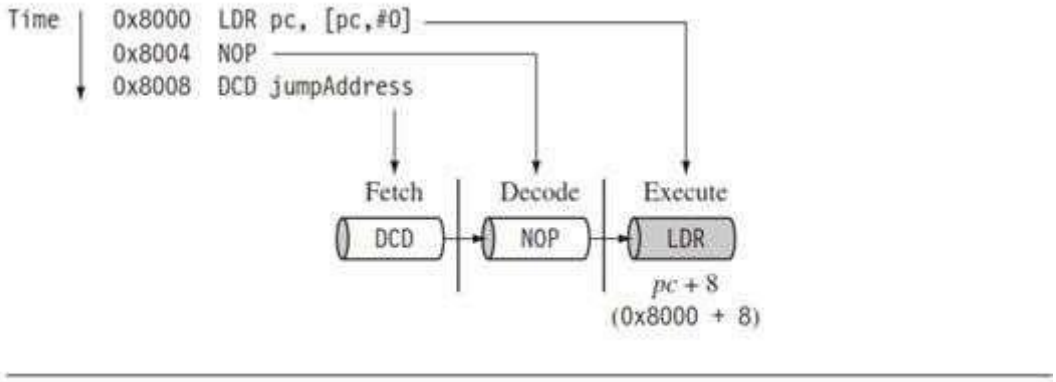


Figure 7.12 Example: pc = address + 8.

- Figure 7.12 illustrates the use of the pipeline and the program counter *pc.*
- In the execute stage, the *pc* always points to the address of the instruction plus 8 bytes.
- In other words, the *pc* always points to the address of the instruction being executed plus two instructions ahead.
- when the *pc* is used for calculating a relative offset and is an architectural characteristic across all the pipelines.

  There are three other characteristics of the pipeline worth mentioning.

  - First, the execution of a branch instruction or branching by the direct modification of the *pc* causes the ARM core to flush its pipeline.

  - Second, ARM10 uses branch prediction, which reduces the effect of a pipeline flush by predicting possible branches and loading the new branch address prior to the execution of the instruction.

  - Third, an instruction in the execute stage will complete even though an interrupt has been raised.

Exceptions, Interrupts, and the Vector Table

Table 7.6 The vector table.

Exception/interrupt	Shorthand	Address	High address
Reset	RESET	0x00000000	0xffff0000
Undefined instruction	UNDEF	0x00000004	0xffff0004
Software interrupt	SWI	0x00000008	0xffff0008
Prefetch abort	PABT	0x0000000c	0xffff000c
Data abort	DABT	0x00000010	0xffff0010
Reserved	—	0x00000014	0xffff0014
Interrupt request	IRQ	0x00000018	0xffff0018
Fast interrupt request	FIQ	0x0000001c	0xffff001c

- When an exception or interrupt occurs, the processor sets the *pc* to a specific memory address.
- Address is within a special address range called the *vector table*.
- Entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
- Memory map address 0x00000000 is reserved for the vector table, a set of 32-bit words.
- On some processors the vector table can be optionally located at a higher address in memory (starting at the offset 0xffff0000).
- Operating systems such as Linux and Microsoft's embedded products can take advantage of this feature.

  ■ *Reset* is *executed* by the processor when power is applied. This instruction branches to the initialization code.

  ■ *Undefined instruction vector* is used when the processor cannot decode an instruction.

  ■ *Software interrupt vector* SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine.

  ■ *Prefetch abort vector* occurs when the processor attempts to fetch an instruction from an address without the correct access permissions. The actual abort occurs in the decode stage.

  ■ *Data abort vector* is similar to a prefetch abort but is raised when an instruction attempts to access data memory without the correct access permissions.

  ■ *Interrupt request vector* is used by external hardware to interrupt the normal execution flow of the processor. It can only be raised if IRQs are not masked in the *cpsr*.

Core Extensions

- Improve performance, manage resources, and provide extra functionality and are designed to provide flexibility in handling particular applications.
- Each ARM family has different extensions available

- Three hardware extensions ARM wraps around the core are:

❑ cache and tightly coupled memory
❑ memory management
❑ coprocessor interface.

Cache and Tightly Coupled Memory

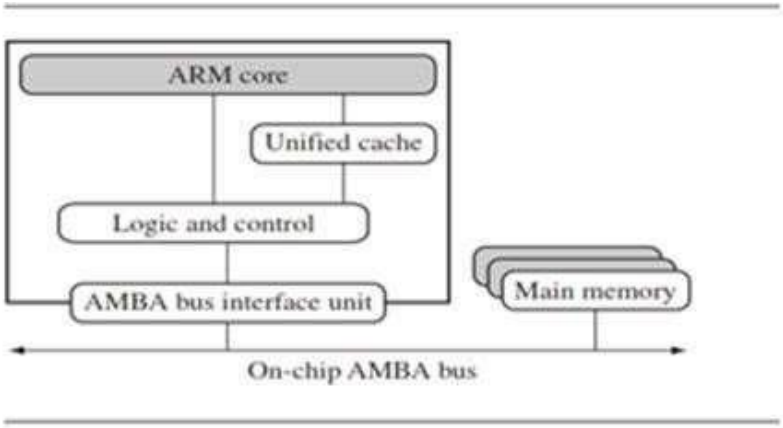


Figure 7.13 A simplified Von Neumann architecture with cache.

- Cache is a block of fast memory placed between main memory and the core.
- With a cache the processor core can run for the majority of the time without having to wait for data from slow external memory.
- single-level cache internal to the processor.
- ARM has two forms of cache. First is found attached to the Von Neumann–style cores.
- It combines both data and instruction into a single unified cache.
- For simplicity, we have called the glue logic that connects the memory system to the AMBA bus *logic and control*
- overall increase in performance but at the expense of predictable execution.
- But for real-time systems it is paramount that code execution is *deterministic*—
- The time taken for loading and storing instructions or data must be predictable.
- This is achieved using a form of memory called *tightly coupled memory* (TCM).
- TCMs appear as memory in the address map and can be accessed as fast memory.
- An example of a processor with TCMs is shown in Figure 2.14.

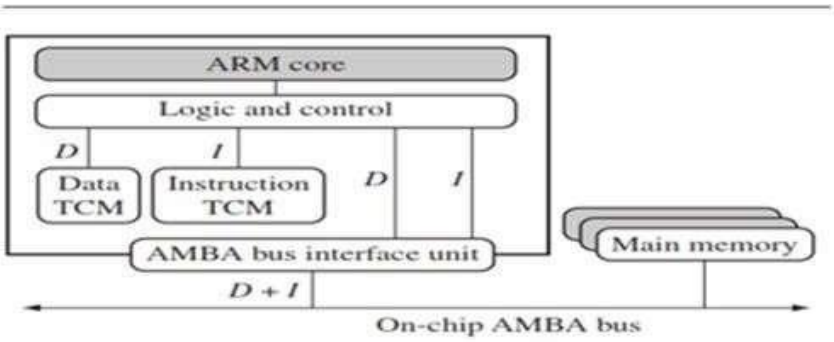


Figure 7.14 A simplified Harvard architecture with TCMs.

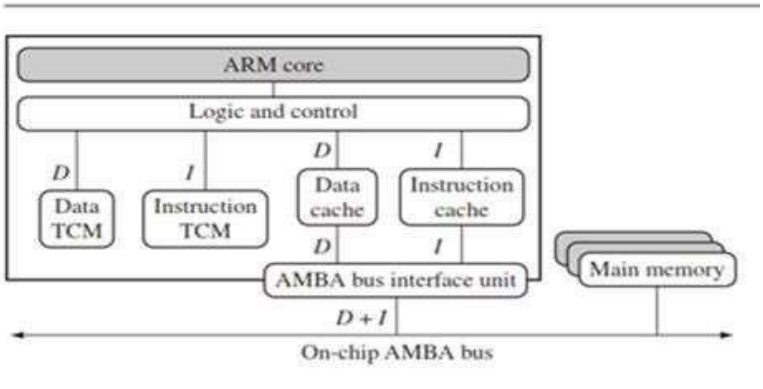


Figure 7.15 A simplified Harvard architecture with caches and TCMs.

- By combining both technologies, ARM processors can have both improved performance and predictable real-time response. Figure 2.15 shows an example core with a combination of caches and TCMs.

Memory Management

- Embedded systems often use multiple memory devices.
- It is usually necessary to have a method to help organize these devices and protect the system from applications trying to make inappropriate accesses to hardware.
- This is achieved with the assistance of memory management hardware.
- ARM cores have three different types of memory
  - no extensions providing no protection,
  - a memory protection unit (MPU) providing limited protection,
  - a memory management unit (MMU) providing full protection:

■ *Nonprotected memory* is fixed and provides very little flexibility.

- It is normally used for small, simple embedded systems that require no protection from rogue applications

■ *MPU*s employ a simple system that uses a limited number of memory regions.

- These regions are controlled with a set of special coprocessor registers,
- each region is defined with specific access permissions.
- This type of memory management is used for systems that require memory protection but don't have a complex memory map.

■ *MMU*s are the most comprehensive memory management hardware available on the ARM.

- The MMU uses a set of translation tables to provide fine-grained control over memory.
- These tables are stored in main memory and provide a virtual-to-physical address map as well as access permissions.
- MMUs are designed for more sophisticated platform operating systems that support multitasking

Coprocessors

- Coprocessors can be attached to the ARM processor.
- More than one coprocessor can be added to the ARM core via the coprocessor interface.
- The coprocessor can be accessed through a group of dedicated ARM instructions that provide a load-store type interface.
- Consider, for example, coprocessor 15: The ARM processor uses coprocessor 15 registers to control the cache, TCMs, and memory management.

****

MODULE -2

ARM INSTRUCTIONS

Introduction to the ARM Instruction Set: Data Processing Instructions , Branch Instructions, Software Interrupt Instructions, Program Status Register Instructions, Coprocessor Instructions, Loading Constants, Simple programming exercises.

The most common and useful ARM instructions are introduced in this module. Different ARM architecture versions support different instructions. But new versions add more instructions and new versions are backward compatible. The following table has a complete list of ARM instructions available in ARMv5E ISA (Instruction Set Architecture). The ARM ISA column in the table lists the ISA revision (version) in which the instruction was introduced.

Table **8.1** ARM instruction set.

Mnemonics	ARM ISA	Description
ADC	v1	add two 32-bit values and carry
ADD	v1	add two 32-bit values
AND	v1	logical bitwise AND of two 32-bit values
B	v1	branch relative +/− 32 MB
BIC	v1	logical bit clear (AND NOT) of two 32-bit values
BKPT	v5	breakpoint instructions
BL	v1	relative branch with link
BLX	v5	branch with link and exchange
BX	v4T	branch with exchange
CDP CDP2	v2 v5	coprocessor data processing operation
CLZ	v5	count leading zeros
CMN	v1	compare negative two 32-bit values
CMP	v1	compare two 32-bit values
EOR	v1	logical exclusive OR of two 32-bit values
LDC LDC2	v2 v5	load to coprocessor single or multiple 32-bit values
LDM	v1	load multiple 32-bit words from memory to ARM registers
LDR	v1 v4 v5E	load a single value from a virtual address in memory
MCR MCR2 MCRR	v2 v5 v5E	move to coprocessor from an ARM register or registers
MLA	v2	multiply and accumulate 32-bit values
MOV	v1	move a 32-bit value into a register
MRC MRC2 MRRC	v2 v5 v5E	move to ARM register or registers from a coprocessor
MRS	v3	move to ARM register from a status register (*cpsr* or *spsr*)
MSR	v3	move to a status register (*cpsr* or *spsr*) from an ARM register
MUL	v2	multiply two 32-bit values
MVN	v1	move the logical NOT of 32-bit value into a register
ORR	v1	logical bitwise OR of two 32-bit values
PLD	v5E	preload hint instruction
QADD	v5E	signed saturated 32-bit add
QDADD	v5E	signed saturated double and 32-bit add
QDSUB	v5E	signed saturated double and 32-bit subtract
QSUB	v5E	signed saturated 32-bit subtract
RSB	v1	reverse subtract of two 32-bit values
RSC	v1	reverse subtract with carry of two 32-bit integers
SBC	v1	subtract with carry of two 32-bit values
SMLA*xy*	v5E	signed multiply accumulate instructions ((16 × 16) + 32 = 32-bit)
SMLAL	v3M	signed multiply accumulate long ((32 × 32) + 64 = 64-bit)
SMLAL*xy*	v5E	signed multiply accumulate long ((16 × 16) + 64 = 64-bit)
SMLAW*y*	v5E	signed multiply accumulate instruction (((32 × 16) ≫ 16) + 32 = 32-bit)
SMULL	v3M	signed multiply long (32 × 32 = 64-bit)

continued

Table **8.1** ARM instruction set. (*Continued*)

Mnemonics	ARM ISA	Description
SMUL*xy*	v5E	signed multiply instructions (16 × 16 = 32-bit)
SMULW*y*	v5E	signed multiply instruction ((32 × 16) ≫ 16 = 32-bit)
STC STC2	v2 v5	store to memory single or multiple 32-bit values from coprocessor
STM	v1	store multiple 32-bit registers to memory
STR	v1 v4 v5E	store register to a virtual address in memory
SUB	v1	subtract two 32-bit values
SWI	v1	software interrupt
SWP	v2a	swap a word/byte in memory with a register, without interruption
TEQ	v1	test for equality of two 32-bit values
TST	v1	test for bits in a 32-bit value
UMLAL	v3M	unsigned multiply accumulate long ((32 × 32) + 64 = 64-bit)
UMULL	v3M	unsigned multiply long (32 × 32 = 64-bit)

The processor operation is illustrated with PRE and POST-conditions. These conditions describe the registers and memory before and after the instruction is executed.
Hexadecimal numbers are represented with the prefix of 0x
Binary numbers are represented with the prefix of 0b.

```
PRE     <pre-conditions>
        <instruction/s>
POST    <post-conditions>
```

In the pre- and post-conditions, memory is denoted as

```
mem<data_size>[address]
```

This refers to data_size - bits of memory starting at the given byte address.
Ex: mem32[1024] is the 32-bit value starting at address 1KB.

ARM instructions process the data present in the registers. Memory can be accessed only with LOAD and STORE instructions. ARM instructions can have 2 or 3 operands.

Ex: The ADD instruction shown below adds the two values stored in the source registers r1 and r2. After adding the sum is stored in the destination register r3.

Instruction Syntax	Destination register (*Rd*)	Source register 1 (*Rn*)	Source register 2 (*Rm*)
ADD r3, r1, r2	*r3*	*r1*	*r2*

Data Processing instructions

The data processing instructions manipulate data within registers. There are 5 types of data processing instructions as listed below:

- Move instructions
- Arithmetic Instructions
- Logical Instructions
- Comparison Instructions
- Multiply Instructions.

Most of the data processing instructions in ARM can process one of their operands using the barrel shifter. Suffix S is used on a data processing instruction, to update the flags in the cpsr.

Move and logical operations update the carry flag *C*, negative flag *N*, and zero flag *Z*.

- The carry flag is set from the result of the barrel shift as the last bit is shifted out.
- The *N* flag is set to bit 31 of the result.
- The *Z* flag is set if the result is zero

2.1.1-MOVE Instructions: Move is the simplest ARM instruction. It copies *N* into a destination register *Rd*, where *N* is a register or immediate value. This instruction is useful for setting initial values and transferring data between registers.

Syntax: <instruction>{<cond>}{S} Rd, N

MOV	Move a 32-bit value into a register	$Rd = N$
MVN	move the NOT of the 32-bit value into a register	$Rd = \sim N$

EXAMPLE 2.1 This example shows a simple move instruction. The MOV instruction takes the contents of register *r5* and copies them into register *r7*, in this case, taking the value 5, and overwriting the value 8 in register *r7*.

```
PRE   r5 = 5
      r7 = 8
      MOV   r7, r5    ; let r7 = r5
POST  r5 = 5
      r7 = 5
```

Barrel Shifter

- In Example 2.1 a MOV instruction is shown where *N* is a simple register. But *N* can be more than just a register or immediate value.
- It can also be a register *Rm* that has been pre-processed by the barrel shifter prior to being used by a data processing instruction.
- Data processing instructions are processed within the arithmetic logic unit (ALU).
- A unique and powerful feature of the ARM processor is the ability to shift the 32-bit binary pattern in one of the source registers left or right by a specific number of positions before it enters the ALU.
- This shift increases the power and flexibility of many data processing operations.
- There are data processing instructions that do not use the barrel shifter, for example, the MUL (multiply), CLZ (count leading zeros), and QADD (signed saturated 32-bit add) instructions.
- Pre-processing or shift occurs within the cycle time of the instruction. This is particularly useful for loading constants into a register and achieving fast multiplies or division by a power of 2.
- To illustrate the barrel shifter add a shift operation to the move instruction example.

- Register *Rn* enters the ALU without any pre-processing of registers. Figure 2.1 shows the data flow between the ALU and the barrel shifter.

EXAMPLE 2.2 We apply a logical shift left (LSL) to register *Rm* before moving it to the destination register. This is the same as applying the standard C language shift operator ≪ to the register. The MOV instruction copies the shift operator result *N* into register *Rd*. *N* represents the result of the LSL operation described in Table 2.2

```
PRE    r5 = 5
       r7 = 8
```

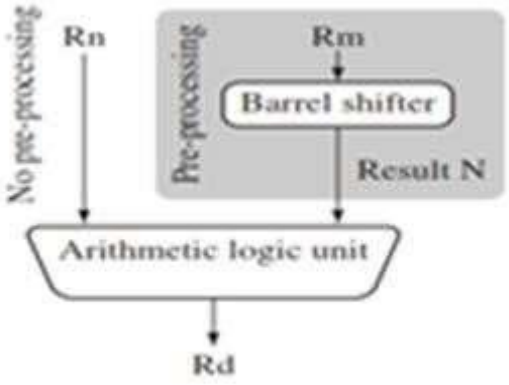


Figure 2.1 Barrel shifter and ALU.

```
       MOV     r7, r5, LSL #2   ; let r7 = r5*4 = (r5<<2)
POST   r5 = 5
       r7 = 20
```

The example multiplies register *r5* by four and then places the result into register *r7*.

- The below diagram illustrates a logical shift left by one. If the contents of bit 0 are shifted to bit 1 then bit 0 is cleared.
- The *C* flag is updated with the last bit shifted out of the register.
- This is bit (32 – *y*) of the original value, where *y* is the shift amount.
- When *y* is greater than one, then a shift by *y* positions is the same as a shift by one position executed *y* times.

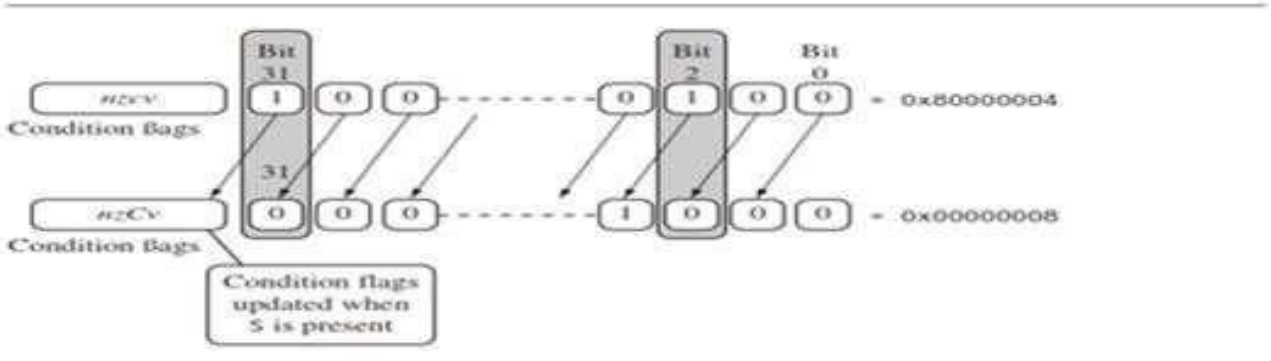


Figure 2.2 Logical shift left by one.

The five different operations that can be used within the barrel shifter are as in the below table.

Table **8.2** Barrel shifter operations.

Mnemonic	Description	Shift	Result	Shift amount y
LSL	logical shift left	*x*LSL *y*	$x \ll y$	#0–31 or *Rs*
LSR	logical shift right	*x*LSR *y*	(unsigned)$x \gg y$	#1–32 or *Rs*
ASR	arithmetic right shift	*x*ASR *y*	(signed)$x \gg y$	#1–32 or *Rs*
ROR	rotate right	*x*ROR *y*	((unsigned)$x \gg y$) | ($x \ll (32 - y)$)	#1–31 or *Rs*
RRX	rotate right extended	*x*RRX	(*c* flag $\ll 31$) | ((unsigned)$x \gg 1$)	none

Note: *x* represents the register being shifted and *y* represents the shift amount.

- Table 3.3 lists the syntax for the different barrel shift operations available on data processing instructions.
- The second operand *N* can be an immediate constant preceded by #, a register value *Rm*, or the value of *Rm* processed by a shift.

Table **8.3** Barrel shift operation syntax for data processing instructions.

N shift operations	Syntax
Immediate	`#immediate`
Register	`Rm`
Logical shift left by immediate	`Rm, LSL #shift_imm`
Logical shift left by register	`Rm, LSL Rs`
Logical shift right by immediate	`Rm, LSR #shift_imm`
Logical shift right with register	`Rm, LSR Rs`
Arithmetic shift right by immediate	`Rm, ASR #shift_imm`
Arithmetic shift right by register	`Rm, ASR Rs`
Rotate right by immediate	`Rm, ROR #shift_imm`
Rotate right by register	`Rm, ROR Rs`
Rotate right with extend	`Rm, RRX`

EXAMPLE **8.3** This example of a MOVS instruction shifts register *r1* left by one bit. This multiplies register *r1* by a value 2^1. As you can see, the *C* flag is updated in the *cpsr* because the S suffix is present in the instruction mnemonic.

```
PRE    cpsr = nzcvqiFt_USER
       r0 = 0x00000000
       r1 = 0x80000004

       MOVS    r0, r1, LSL #1

POST   cpsr = nzCvqiFt_USER
       r0 = 0x00000008
       r1 = 0x80000004
```

ARITHMETIC instructions

- The arithmetic instructions implement addition and subtraction of 32-bit signed and unsigned values.

Syntax: <instruction>{<cond>}{S} Rd, Rn, N

ADC	add two 32-bit values and carry	$Rd = Rn + N +$ carry
ADD	add two 32-bit values	$Rd = Rn + N$
RSB	reverse subtract of two 32-bit values	$Rd = N - Rn$
RSC	reverse subtract with carry of two 32-bit values	$Rd = N - Rn - !$(carry flag)
SBC	subtract with carry of two 32-bit values	$Rd = Rn - N - !$(carry flag)
SUB	subtract two 32-bit values	$Rd = Rn - N$

N is the result of the shifter operation. The syntax of shifter operation is shown in Table 3.3.

EXAMPLE **8.4** This simple subtract instruction subtracts a value stored in register *r2* from a value stored in register *r1*. The result is stored in register *r0*.

```
PRE   r0 = 0x00000000
      r1 = 0x00000002
      r2 = 0x00000001

      SUB r0, r1, r2

POST  r0 = 0x00000001
```

EXAMPLE **8.5** This reverse subtract instruction (RSB) subtracts *r1* from the constant value #0, writing the result to *r0*. You can use this instruction to negate numbers.

```
PRE   r0 = 0x00000000
      r1 = 0x00000077

      RSB r0, r1, #0      ; Rd = 0x0 - r1

POST  r0 = -r1 = 0xffffff89
```

EXAMPLE **8.6** The SUBS instruction is useful for decrementing loop counters. In this example we subtract the immediate value one from the value one stored in register *r1*. The result value zero is written to register *r1*. The *cpsr* is updated with the *ZC* flags being set.

```
PRE   cpsr = nzcvqiFt_USER
      r1 = 0x00000001

      SUBS r1, r1, #1

POST  cpsr = nZCvqiFt_USER
      r1 = 0x00000000
```

Using the Barrel Shifter with Arithmetic Instructions

- The wide range of second operand shifts available on arithmetic and logical instructions is a very powerful feature of the ARM instruction set.
- Example 3.7 illustrates the use of the inline barrel shifter with an arithmetic instruction.
- The instruction multiplies the value stored in register *r1* by three.

EXAMPLE 8.7 Register *r1* is first shifted one location to the left to give the value of twice *r1*. The ADD instruction then adds the result of the barrel shift operation to register *r1*. The final result transferred into register *r0* is equal to three times the value stored in register *r1*.

```
PRE   r0 = 0x00000000
      r1 = 0x00000005

      ADD    r0, r1, r1, LSL #1

POST  r0 = 0x0000000f
      r1 = 0x00000005
```

Logical Instructions

- Logical instructions perform bitwise logical operations on the two source registers.

```
Syntax: <instruction>{<cond>}{S} Rd, Rn, N
```

AND	logical bitwise AND of two 32-bit values	$Rd = Rn \& N$
ORR	logical bitwise OR of two 32-bit values	$Rd = Rn \mid N$
EOR	logical exclusive OR of two 32-bit values	$Rd = Rn \wedge N$
BIC	logical bit clear (AND NOT)	$Rd = Rn \& \sim N$

EXAMPLE 8.8 This example shows a logical OR operation between registers *r1* and *r2*. *r0* holds the result.

```
PRE   r0 = 0x00000000
      r1 = 0x02040608
      r2 = 0x10305070

      ORR   r0, r1, r2

POST  r0 = 0x12345678
```

EXAMPLE 8.9 This example shows a more complicated logical instruction called BIC, which carries out a logical bit clear.

```
PRE   r1 = 0b1111
      r2 = 0b0101

      BIC   r0, r1, r2

POST  r0 = 0b1010
```

This is equivalent to `Rd = Rn AND NOT(N)`

- In this example, register *r2* contains a binary pattern, where every binary 1 in *r2* clears a corresponding bit location in register *r1*.
- This instruction is particularly useful when clearing status bits and is frequently used to change interrupt masks in the *cpsr*.
- The logical instructions update the *cpsr* flags only if the S suffix is present.
- These instructions can use barrel-shifted second operands in the same way as the arithmetic instructions.

Comparison Instructions

- The comparison instructions are used to compare or test a register with a 32-bit value.
- They update the *cpsr* flag bits according to the result, but do not affect other registers.
- After the bits have been set, the information can then be used to change program flow by using conditional execution.
- S suffix is not necessary for comparison instructions to update the flags.

```
Syntax: <instruction>{<cond>} Rn, N
```

CMN	compare negated	flags set as a result of $Rn + N$
CMP	compare	flags set as a result of $Rn - N$
TEQ	test for equality of two 32-bit values	flags set as a result of $Rn \wedge N$
TST	test bits of a 32-bit value	flags set as a result of $Rn \& N$

- The CMP is effectively a subtract instruction with the result discarded; similarly the TST instruction is a logical AND operation, and TEQ is a logical exclusive OR operation.
- For each, the results are discarded but the condition bits are updated in the *cpsr*. It is important to understand that comparison instructions only modify the conditional flags of the *cpsr* and do not affect the registers being compared.

EXAMPLE **8.10** This example shows a CMP comparison instruction. You can see that both registers, *r0* and *r9*, are equal before executing the instruction. The value of the *z* flag prior to execution is 0 and is represented by a lowercase *z*. After execution the *z* flag changes to 1 or an uppercase *Z*. This change indicates *equality*.

```
PRE    cpsr = nzcvqiFt_USER
       r0 = 4
       r9 = 4

       CMP   r0, r9

POST   cpsr = nZcvqiFt_USER
```

Multiply Instructions

- The multiply instructions multiply the contents of a pair of registers and, depending upon the instruction, accumulate the results in with another register.
- The long multiplies accumulate onto a pair of registers representing a 64-bit value.
- The final result is placed in a destination register or a pair of registers.

```
Syntax: MLA{<cond>}{S} Rd, Rm, Rs, Rn
        MUL{<cond>}{S} Rd, Rm, Rs
```

MLA	multiply and accumulate	$Rd = (Rm * Rs) + Rn$
MUL	multiply	$Rd = Rm * Rs$

```
Syntax: <instruction>{<cond>}{S} RdLo, RdHi, Rm, Rs
```

SMLAL	signed multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
SMULL	signed multiply long	$[RdHi, RdLo] = Rm * Rs$
UMLAL	unsigned multiply accumulate long	$[RdHi, RdLo] = [RdHi, RdLo] + (Rm * Rs)$
UMULL	unsigned multiply long	$[RdHi, RdLo] = Rm * Rs$

- The number of cycles taken to execute a multiply instruction depends on the processor implementation. For some implementations the cycle timing also depends on the value in *Rs*.

EXAMPLE **8.11** This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value 2, and *r2* is equal to 2. The result, 4, is then placed into register *r0*.

```
PRE   r0 = 0x00000000
      r1 = 0x00000002
      r2 = 0x00000002

      MUL   r0, r1, r2   ; r0 = r1*r2

POST  r0 = 0x00000004
      r1 = 0x00000002
      r2 = 0x00000002
```

- The long multiply instructions (SMLAL, SMULL, UMLAL, and UMULL) produce a 64-bit result.
- If the result is too large to fit in a single 32-bit register, then the result is placed in two registers labeled *RdLo* and *RdHi*. *RdLo* holds the lower 32 bits of the 64-bit result, and *RdHi* holds the higher 32 bits of the 64-bit result.
- Example 8.12 shows an example of a long unsigned multiply instruction.

EXAMPLE **8.12** This example shows a simple multiply instruction that multiplies registers *r1* and *r2* together and places the result into register *r0*. In this example, register *r1* is equal to the value 2, and *r2* is equal to 2. The result, 4, is then placed into register *r0*.

```
PRE   r0 = 0x00000000
      r1 = 0x00000000
      r2 = 0xf0000002
      r3 = 0x00000002
    UMULL   r0, r1, r2,r3   ; [r1,r0] = r2 * r3

POST  r0 = 0xe0000004   ; = RdLo
      r1 = 0x00000001   ; = RdHi
```

Branch Instructions

- A branch instruction changes the flow of execution or is used to call a routine. This type of instruction allows programs to have subroutines, *if-then-else* structures, and loops.
- The change of execution flow forces the program counter *pc* to point to a new address. The ARMv5E instruction set includes four different branch instructions.

```
Syntax: B{<cond>} label
        BL{<cond>} label
        BX{<cond>} Rm
        BLX{<cond>} label | Rm
```

B	branch	*pc* = *label*
BL	branch with link	*pc* = *label* *lr* = address of the next instruction after the BL
BX	branch exchange	*pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1
BLX	branch exchange with link	*pc* = *label*, *T* = 1 *pc* = *Rm* & 0xfffffffe, *T* = *Rm* & 1 *lr* = address of the next instruction after the BLX

- The address *label* is stored in the instruction as a signed *pc*-relative offset and must be within approximately 32 MB of the branch instruction.
- *T* refers to the Thumb bit in the *cpsr*. When instructions set *T*, the ARM switches to Thumb state.

EXAMPLE **8.13** This example shows a forward and backward branch. Because these loops are address specific, we do not include the pre- and post-conditions. The forward branch skips three instructions. The backward branch creates an infinite loop.

```
        B       forward
        ADD     r1, r2, #4
        ADD     r0, r6, #2
        ADD     r3, r7, #4
forward
        SUB     r1, r2, #4
```

```
backward
        ADD     r1, r2, #4
        SUB     r1, r2, #4
        ADD     r4, r6, r7
        B       backward
```

- Branches are used to change execution flow. Most assemblers hide the details of a branch instruction encoding by using labels.
- In the above example, *forward* and *backward* are the labels.
- The branch labels are placed at the beginning of the line and are used to mark an address that can be used later by the assembler to calculate the branch offset.

EXAMPLE **8.14** The branch with link, or BL, instruction is similar to the B instruction but overwrites the link register *lr* with a return address. It performs a subroutine call. This example shows a simple fragment of code that branches to a subroutine using the BL instruction. To return from a subroutine, you copy the link register to the *pc*.

```
        BL      subroutine      ; branch to subroutine
        CMP     r1, #5          ; compare r1 with 5
        MOVEQ   r1, #0          ; if (r1==5) then r1 = 0
        :
subroutine
        <subroutine code>
        MOV     pc, lr          ; return by moving pc = lr
```

The branch exchange (BX) instruction uses an absolute address stored in register Rm. It is mainly used to branch to and from the Thumb code. The T bit of cpsr is updated by the LSB of the branch register. Similarly the BLX instruction updates the T bit of the cpsr with LSB and also sets the link register with the return address.

LOAD-STORE INSTRUCTIONS

Load-store instructions transfer data between memory and processor registers. There are three types of load-store instructions:

- Single-register Transfer.
- Multiple-register Transfer.
- Swap

Single-Register Transfer: These instructions are used for moving a single data item in and out of a register. The data types supported are

- Signed And Unsigned Words (32-bit).
- Halfwords (16-bit).
- Bytes.

A few load-store single-register transfer instructions are shown below:

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing1
        LDR{<cond>}SB|H|SH Rd, addressing2
        STR{<cond>}H Rd, addressing2
```

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*

LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

EXAMPLE **8.15** LDR and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored. For example, LDR can only load 32-bit words on a memory address that is a multiple of four bytes—0, 4, 8, and so on. This example shows a load from a memory address contained in register *r1*, followed by a store back to the same address in memory.

```
;
; load register r0 with the contents of
; the memory address pointed to by register
; r1.
;
        LDR     r0, [r1]          ; = LDR r0, [r1, #0]
;
; store the contents of register r0 to
; the memory address pointed to by
; register r1.
;
        STR     r0, [r1]          ; = STR r0, [r1, #0]
```

The first instruction loads a word from the address stored in register *r1* and places it into register *r0*. The second instruction goes the other way by storing the contents of register *r0* to the address contained in register *r1*. The offset from register *r1* is zero. Register *r1* is called the *base address register*.

Single-Register Load-Store Addressing Modes: The ARM instruction set provides different modes for addressing memory. These modes incorporate one of the indexing methods:

- Preindex With Writeback.
- Preindex.
- Postindex.

Table **8.4** Index methods.

Index method	Data	Base address register	Example
Preindex with writeback	*mem[base + offset]*	*base + offset*	LDR r0,[r1,#4]!
Preindex	*mem[base + offset]*	*not updated*	LDR r0,[r1,#4]
Postindex	*mem[base]*	*base + offset*	LDR r0,[r1],#4

Note: ! indicates that the instruction writes the calculated address back to the base address register.

EXAMPLE **8.16** Preindex with writeback calculates an address from a base register plus address offset and then updates that address base register with the new address. In contrast, the preindex offset is the same as the preindex with writeback but does not update the address base register. Postindex only updates the address base register after the address is used. The preindex mode is useful for accessing an element in a data structure. The postindex and preindex with writeback modes are useful for traversing an array.

```
PRE     r0 = 0x00000000
        r1 = 0x00090000
        mem32[0x00009000] = 0x01010101
        mem32[0x00009004] = 0x02020202

        LDR     r0, [r1, #4]!
```

Preindexing with writeback:

```
POST(1) r0 = 0x02020202
        r1 = 0x00009004

        LDR     r0, [r1, #4]
```

Preindexing:

```
POST(2) r0 = 0x02020202
        r1 = 0x00009000

        LDR     r0, [r1], #4
```

Postindexing:

```
POST(3) r0 = 0x01010101
        r1 = 0x00009004
```

Table **8.5** Single-register load-store addressing, word or unsigned byte.

Addressing[1] mode and index method	Addressing[1] syntax
Preindex with immediate offset	[Rn, #+/-offset_12]
Preindex with register offset	[Rn, +/-Rm]
Preindex with scaled register offset	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with scaled register offset	[Rn, +/-Rm, shift #shift_imm]!
Immediate postindexed	[Rn], #+/-offset_12
Register postindex	[Rn], +/-Rm
Scaled register postindex	[Rn], +/-Rm, shift #shift_imm

- Example 8.15 used a pre index method. This example shows how each indexing method effects the address held in register *r1*, as well as the data loaded into register *r0*.
- Each instruction shows the result of the index method with the same pre-condition.
- The addressing modes available with a particular load or store instruction depend on the instruction class.
- Table 8.5 shows the addressing modes available for load and store of a 32-bit word or an unsigned byte.
- A signed offset or register is denoted by "+/−", identifying that it is either a positive or negative offset from the base address register *Rn*. The base address register is a pointer to a byte in memory, and the offset specifies a number of bytes.
- *Immediate* means the address is calculated using the base address register and a 12-bit offset encoded in the instruction.
- *Register* means the address is calculated using the base address register and a specific register's contents.
- *Scaled* means the address is calculated using the base address register and a barrel shift operation.
- Table 8.6 provides an example of the different variations of the LDR instruction.
- Table 8.7 shows the addressing modes available on load and store instructions using 16-bit halfword or signed byte data. These operations cannot use the barrel shifter. There are no STRSB or STRSH instructions since STRH stores both a signed and unsigned halfword; similarly STRB stores signed and unsigned bytes.
- Table 8.8 shows the variations for STRH instructions.

Table **8.6** Examples of LDR instructions using different addressing modes.

	Instruction	*r0* =	*r1* + =
Preindex with writeback	`LDR r0,[r1,#0x4]!`	`mem32[r1+0x4]`	`0x4`
	`LDR r0,[r1,r2]!`	`mem32[r1+r2]`	`r2`
	`LDR r0,[r1,r2,LSR#0x4]!`	`mem32[r1+(r2 LSR 0x4)]`	`(r2 LSR 0x4)`
Preindex	`LDR r0,[r1,#0x4]`	`mem32[r1+0x4]`	*not updated*
	`LDR r0,[r1,r2]`	`mem32[r1+r2]`	*not updated*
	`LDR r0,[r1,-r2,LSR #0x4]`	`mem32[r1-(r2 LSR 0x4)]`	*not updated*
Postindex	`LDR r0,[r1],#0x4`	`mem32[r1]`	`0x4`
	`LDR r0,[r1],r2`	`mem32[r1]`	`r2`
	`LDR r0,[r1],r2,LSR #0x4`	`mem32[r1]`	`(r2 LSR 0x4)`

Table **8.7** Single-register load-store addressing, halfword, signed halfword, signed byte, and doubleword.

Addressing[2] mode and index method	Addressing[2] syntax
Preindex immediate offset	`[Rn, #+/-offset_8]`
Preindex register offset	`[Rn, +/-Rm]`
Preindex writeback immediate offset	`[Rn, #+/-offset_8]!`
Preindex writeback register offset	`[Rn, +/-Rm]!`
Immediate postindexed	`[Rn], #+/-offset_8`
Register postindexed	`[Rn], +/-Rm`

Table **8.8** Variations of STRH instructions.

	Instruction	Result	*r1* + =
Preindex with writeback	`STRH r0,[r1,#0x4]!`	`mem16[r1+0x4]=r0`	`0x4`
	`STRH r0,[r1,r2]!`	`mem16[r1+r2]=r0`	`r2`
Preindex	`STRH r0,[r1,#0x4]`	`mem16[r1+0x4]=r0`	*not updated*
	`STRH r0,[r1,r2]`	`mem16[r1+r2]=r0`	*not updated*
Postindex	`STRH r0,[r1],#0x4`	`mem16[r1]=r0`	`0x4`
	`STRH r0,[r1],r2`	`mem16[r1]=r0`	`r2`

Multiple-Register Transfer

- Load-store multiple instructions can transfer multiple registers between memory and the processor in a single instruction. The transfer occurs from a base address register *Rn* pointing into memory.
- Multiple-register transfer instructions are more efficient than single-register transfers for moving blocks of data around memory and saving and restoring context and stacks.
- Load-store multiple instructions can increase interrupt latency. ARM implementations do not usually accept interrupt instructions while they are executing.
- For example, on an ARM7 a load multiple instruction takes 2 + *Nt* cycles, where *N* is the number of registers to load, *t* is the number of cycles required for each sequential access to memory.
- If an interrupt has been raised, then it has no effect until the load-store multiple instruction is complete. Compilers, such as armcc, provide a switch to control the

maximum number of registers being transferred on a load-store, which limits the maximum interrupt latency.

```
Syntax: <LDM|STM>{<cond>}<addressing mode> Rn{!},<registers>{^}
```

LDM	load multiple registers	{Rd}*N <- mem32[start address + 4*N] optional Rn updated
STM	save multiple registers	{Rd}*N -> mem32[start address + 4*N] optional Rn updated

In table 8.9 N is the number of registers in the list of registers. Any subset of the current bank of registers can be transferred to memory or fetched from memory.The base register Rn determines the source or destination address for a load-store multiple instruction. This register can be optionally updated following the transfer. This occurs when register Rn is followed by the '!' character, similar to the single register load-store using pre-index with writeback.

Table **8.9** Addressing mode for load-store multiple instructions.

Addressing mode	Description	Start address	End address	*Rn*!
IA	increment after	*Rn*	*Rn* + 4*N − 4	*Rn* + 4*N
IB	increment before	*Rn* + 4	*Rn* + 4*N	*Rn* + 4*N
DA	decrement after	*Rn* − 4*N + 4	*Rn*	*Rn* − 4*N
DB	decrement before	*Rn* − 4*N	*Rn* − 4	*Rn* − 4*N

EXAMPLE **8.17** In this example, register *r0* is the base register *Rn* and is followed by !, indicating that the register is updated after the instruction is executed. You will notice within the load multiple instruction that the registers are not individually listed. Instead the "-" character is used to identify a range of registers. In this case the range is from register *r1* to *r3* inclusive.

Each register can also be listed, using a comma to separate each register within "{" and "}" brackets.

```
PRE     mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02

        mem32[0x80010] = 0x01
        r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000

        LDMIA    r0!, {r1-r3}

POST    r0 = 0x0008001c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003
```

Fig.8.3 shows a graphical representation. The Base register r0 points to memory address 0x80010 in the PRE-condition. Memory addresses 0x80010, 0x80014, and 0x80018 contain the values 1,2, and 3 respectively. After the load multiple instruction executes registers r1, r2, and r3 contain these values as shown in 8.4. The base register r0 now points to memory address 0x8001c after the last loaded word.

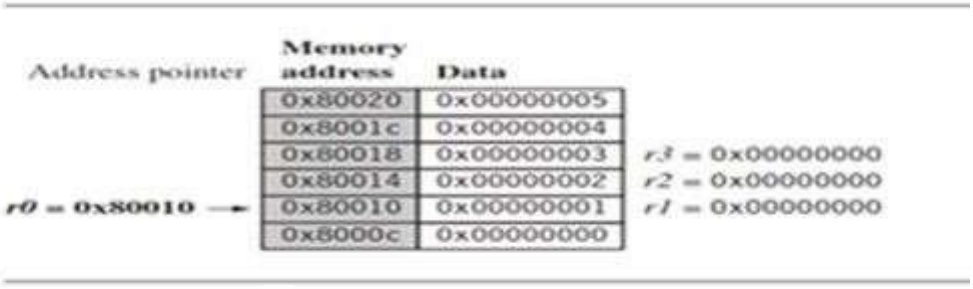


Figure 8.3 Pre-condition for LDMIA instruction.

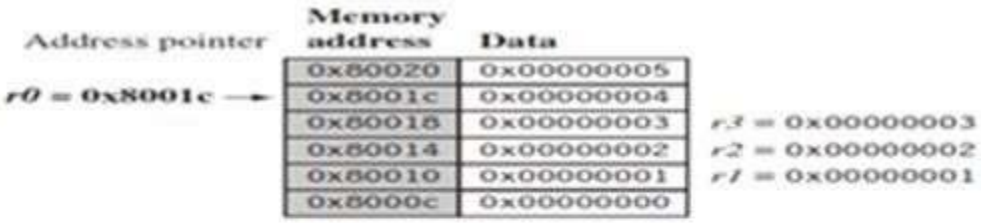


Figure 8.4 Post-condition for LDMIA instruction.

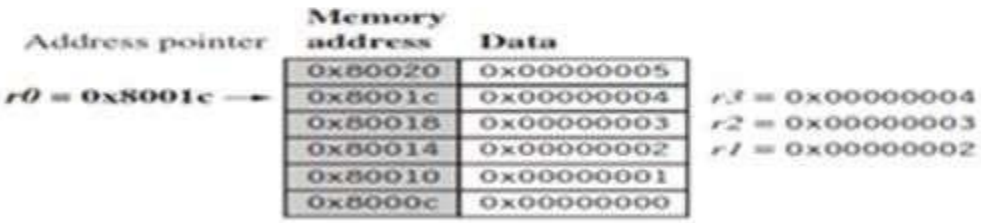


Figure 8.5 Post-condition for LDMIB instruction.

For the same pre-conditions, use LDMIB-Load Multiple Increment before. The first word pointed by register r0 is ignored and register r1 is loaded from the next memory location as shown in fig-3.5. After execution, register r0 now points to the last loaded memory location. This is opposite to LDMIA example, which pointed to the next memory location.

Table 8.10 Load-store multiple pairs when base update used.

Store multiple	Load multiple
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Table 8.10 shows a list of load-store multiple instruction pairs. If a store is used with base update, then the paired load instruction of the same number of registers will reload the data and restore the base address pointer. This is useful to store a group of registers temporarily and restore them later.

EXAMPLE **8.18** This example shows an STM *increment before* instruction followed by an LDM *decrement after* instruction.

```
PRE     r0 = 0x00009000
        r1 = 0x00000009
        r2 = 0x00000008
        r3 = 0x00000007

        STMIB   r0!, {r1-r3}

        MOV     r1, #1
        MOV     r2, #2
        MOV     r3, #3

PRE(2)  r0 = 0x0000900c
        r1 = 0x00000001
        r2 = 0x00000002
        r3 = 0x00000003

        LDMDA r0!, {r1-r3}

POST    r0 = 0x00009000
        r1 = 0x00000009
        r2 = 0x00000008
        r3 = 0x00000007
```

The STMIB instruction stores the values 7, 8, 9 to memory. We then corrupt register *r1* to *r3*. The LDMDA reloads the original values and restores the base pointer *r0*.

EXAMPLE **8.19** We illustrate the use of the load-store multiple instructions with a *block memory copy* example. This example is a simple routine that copies blocks of 32 bytes from a source address location to a destination address location.

The example has two load-store multiple instructions, which use the same *increment after* addressing mode.

```
; r9 points to start of source data
; r10 points to start of destination data
; r11 points to end of the source

loop
        ; load 32 bytes from source and update r9 pointer
        LDMIA   r9!, {r0-r7}



; store 32 bytes to destination and update r10 pointer
STMIA   r10!, {r0-r7} ; and store them

; have we reached the end
CMP     r9, r11
BNE     loop
```

This routine relies on registers r9, r10, and r11 being set up before the code is executed. Registers r9 and r11 determine the data to be copied, and register r10 points to the destination in memory for the data. LDMIA loads the data pointed to by register r9 into registers r0 to r7. It also updates r9 to point to the next block of data to be copied.

STMIA copies the contents of registers r0 to r7 to the destination memory address pointed to by register r10. It also updates r10 to point to the next destination location. CMP and BNE compare pointers r9 and r11 to check whether the end of the block copy has been reached. If the block copy is complete, then the routine

finishes; else the loop repeats with the updated values of register r9 and r10. BNE is the branch instruction B with a condition mnemonic NE (not equal).

Fig-8.6 shows the memory map of the block memory copy and how the routine moves through memory. This loop can transfer 32 bytes i.e. 8 words in two instructions, for a maximum possible throughput of 46 MB/second being transferred at 33 Mhz.

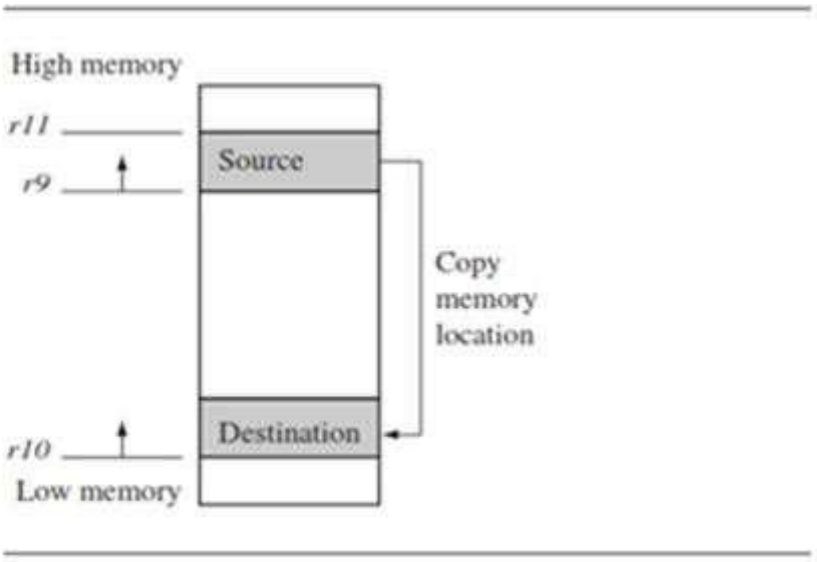


Figure **8.6** Block memory copy in the memory map.

Stack Operations

- The ARM architecture uses the load-store multiple instructions to carry out stack operations.
- The *pop* operation (removing data from stack) uses a load multiple instruction; similarly, the *push* operation (placing data onto the stack) uses a store multiple instruction.
- When using a stack you have to decide whether the stack will grow up or down in memory. A stack is either *ascending* (A) or *descending* (D). Ascending stacks grow towards higher memory addresses; in contrast, descending stacks grow towards lower memory addresses.
- When you use a *full stack* (F), the stack pointer *sp* points to an address that is the last used or full location (i.e., *sp* points to the last item on the stack).
- In contrast, if you use an *empty stack* (E) the *sp* points to an address that is the first unused or empty location (i.e., it points after the last item on the stack).

EXAMPLE **8.20** The STMFD instruction pushes registers onto the stack, updating the *sp*. Figure **8.7** shows a push onto a full descending stack. You can see that when the stack grows the stack pointer points to the last full entry in the stack.

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014

        STMFD   sp!, {r1,r4}
```

Table **8.11** Addressing methods for stack operations.

Addressing mode	Description	Pop	= LDM	Push	= STM
FA	full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	full descending	LDMFD	LDMIA	STMFD	STMDB
EA	empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	empty descending	LDMED	LDMIB	STMED	STMDA

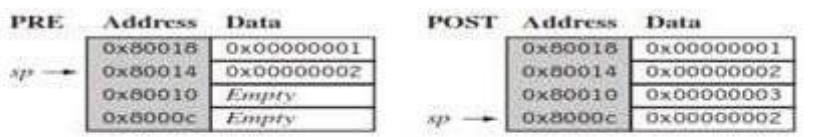


Figure **8.7** STMFD instruction—full stack push operation.

```
POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x0008000c
```

EXAMPLE **8.21** In contrast, Figure 3.8 shows a push operation on an empty stack using the STMED instruction. The STMED instruction pushes the registers onto the stack but updates register *sp* to point to the next empty location.

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080010

        STMED   sp!, {r1,r4}

POST    r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080008
```

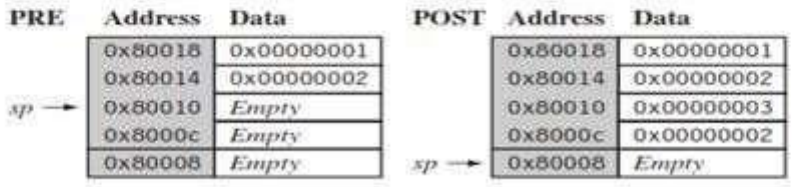


Figure **8.8** STMED instruction—empty stack push operation.

When handling a checked stack 3 attributes have to be preserved, i.e. the stack base, the stack pointer, and the stack limit. Stack base is the starting address of the stack in memory. Stack pointer initially points to the stack base; as data is pushed onto the stack, the stack pointer descends memory and continuously pointsto the top of stack. If the stack pointer passes the stack limit, then a stack overflow error occurs.
Ex: To check for stack overflow

```
SUB sp, sp, #size
CMP sp, r10
BLLO_stack_overflow; condition.
```

SWAP INSTRUCTION

- The swap instruction is a special case of a load-store instruction. It swaps the contents of memory with the contents of a register.
- This instruction is an *atomic operation*—it reads and writes a location in the same bus operation, preventing any other instruction from reading or writing to that location until it completes.
- This instruction is mainly useful for implementing semaphores and mutual exclusion in an operating system. It allows both a byte and a word swap.

```
Syntax: SWP{B}{<cond>} Rd,Rm,[Rn]
```

SWP	swap a word between memory and a register	*tmp = mem32[Rn]* *mem32[Rn] = Rm* *Rd = tmp*
SWPB	swap a byte between memory and a register	*tmp = mem8[Rn]* *mem8[Rn] = Rm* *Rd = tmp*

Swap cannot be interrupted by any other instruction or any other bus access. We say the system "holds the bus" until the transaction is complete.

EXAMPLE **8.22** The swap instruction loads a word from memory into register *r0* and overwrites the memory with register *r1*.

```
PRE   mem32[0x9000] = 0x12345678
      r0 = 0x00000000
      r1 = 0x11112222
      r2 = 0x00009000

      SWP   r0, r1, [r2]

POST  mem32[0x9000] = 0x11112222
      r0 = 0x12345678
      r1 = 0x11112222
      r2 = 0x00009000
```

SOFTWARE INTERRUPT INSTRUCTION

A software interrupt instruction (SWI) causes a software interrupt exception, which provides a mechanism for applications to call operating system routines.

```
Syntax: SWI{<cond>} SWI_number
```

SWI	software interrupt	*lr_svc* = address of instruction following the SWI *spsr_svc* = *cpsr* *pc* = vectors + 0x8 *cpsr* mode = *SVC* *cpsr I* = 1 (mask IRQ interrupts)

- When the processor executes an SWI instruction, it sets the program counter *pc* to the offset 0x8 in the vector table. The instruction also forces the processor mode to *SVC*, which allows an operating system routine to be called in a privileged mode.
- Each SWI instruction has an associated SWI number, which is used to represent a particular function call or feature

EXAMPLE 8.24 Here we have a simple example of an SWI call with SWI number 0x123456, used by ARM toolkits as a debugging SWI. Typically the SWI instruction is executed in user mode.

```
PRE    cpsr = nzcVqift_USER
       pc = 0x00008000
       lr = 0x003fffff; lr = r14
       r0 = 0x12

     0x00008000   SWI     0x123456

POST   cpsr = nzcVqIft_SVC
       spsr = nzcVqift_USER
       pc = 0x00000008
       lr = 0x00008004
       r0 = 0x12
```

- Since SWI instructions are used to call operating system routines, you need some form of parameter passing. This is achieved using registers.
- In this example, register *r0* is used to pass the parameter 0x12. The return values are also passed back via registers. Code called the *SWI handler* is required to process the SWI call.
- The handler obtains the SWI number using the address of the executed instruction, which is calculated from the link register *lr*.
- The SWI number is determined by SWI_Number = <SWI instruction> AND NOT (0xff 000000). Here the *SWI instruction* is the actual 32-bit SWI instruction executed by the processor.

EXAMPLE 8.25 This example shows the start of an SWI handler implementation. The code fragment determines what SWI number is being called and places that number into register *r10*. You can see from this example that the load instruction first copies the complete SWI instruction into register *r10*. The BIC instruction masks off the top bits of the instruction, leaving the SWI number. We assume the SWI has been called from ARM state.

```
SWI_handler
        ;
        ; Store registers r0-r12 and the link register
        ;
        STMFD   sp!, {r0-r12, lr}
        ; Read the SWI instruction
        LDR       r10, [lr, #-4]
        ; Mask off top 8 bits
        BIC       r10, r10, #0xff000000
        ; r10 - contains the SWI number
        BL        service_routine
        ; return from SWI handler
        LDMFD   sp!, {r0-r12, pc}^
```

The number in register *r10* is then used by the SWI handler to call the appropriate SWI service routine.

PROGRAM STATUS REGISTER INSTRUCTIONS

- The ARM instruction set provides two instructions to directly control a program status register (*psr*).
- The MRS instruction transfers the contents of either the *cpsr* or *spsr* into a register; in the reverse direction.
- The MSR instruction transfers the contents of a register into the *cpsr* or *spsr*.
- Together these instructions are used to read and write the *cpsr* and *spsr*.
- The syntax has a label called *fields*. This can be any combination of control (*c*), extension (*x*), status (*s*), and flags (*f*).
- These fields relate to particular byte regions in a *psr*, as shown in Figure 3.9.

```
Syntax: MRS{<cond>} Rd,<cpsr|spsr>
        MSR{<cond>} <cpsr|spsr>_<fields>,Rm
        MSR{<cond>} <cpsr|spsr>_<fields>,#immediate
```

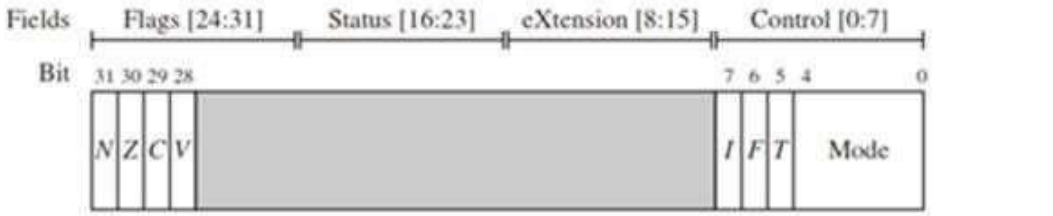


Figure 8.9 *psr* byte fields.

MRS	copy program status register to a general-purpose register	*Rd = psr*
MSR	move a general-purpose register to a program status register	*psr[field] = Rm*
MSR	move an immediate value to a program status register	*psr[field] = immediate*

The *c* field controls the interrupt masks, Thumb state, and processor mode. Example 3.26 shows how to enable IRQ interrupts by clearing the *I* mask. This operation involves using both the MRS and MSR instructions to read from and then write to the *cpsr*.

EXAMPLE **8.26** The MSR first copies the *cpsr* into register *r1*. The BIC instruction clears bit 7 of *r1*. Register *r1* is then copied back into the *cpsr*, which enables IRQ interrupts. You can see from this example that this code preserves all the other settings in the *cpsr* and only modifies the *I* bit in the control field.

```
PRE    cpsr = nzcvqIFt_SVC

       MRS    r1, cpsr
       BIC    r1, r1, #0x80 ; 0b01000000
       MSR    cpsr_c, r1

POST   cpsr = nzcvqiFt_SVC
```

This example is in *SVC* mode. In *user* mode you can read all *cpsr* bits, but you can only update the condition flag field *f*.

Coprocessor Instructions

- Coprocessor instructions are used to extend the instruction set.
- A coprocessor can either provide additional computation capability or be used to control the memory subsystem including caches and memory management.
- The coprocessor instructions include data processing, register transfer, and memory transfer instructions.

```
Syntax: CDP{<cond>} cp, opcode1, Cd, Cn {, opcode2}
        <MRC|MCR>{<cond>} cp, opcode1, Rd, Cn, Cm {, opcode2}
        <LDC|STC>{<cond>} cp, Cd, addressing
```

CDP	coprocessor data processing—perform an operation in a coprocessor
MRC MCR	coprocessor register transfer—move data to/from coprocessor registers
LDC STC	coprocessor memory transfer—load and store blocks of memory to/from a coprocessor

- In the syntax of the coprocessor instructions, the *cp* field represents the coprocessor number between *p0* and *p15*.
- The *opcode* fields describe the operation to take place on the coprocessor.
- The *Cn*, *Cm*, and *Cd* fields describe registers within the coprocessor.
- The coprocessor operations and registers depend on the specific coprocessor you are using. Coprocessor 15 (CP15) is reserved for system control purposes, such as memory management, write buffer control, cache control, and identification registers.

EXAMPLE **8.27** This example shows a CP15 register being copied into a general-purpose register.

```
; transferring the contents of CP15 register c0 to register r10
        MRC p15, 0, r10, c0, c0, 0
```

Here CP15 *register-0* contains the processor identification number. This register is copied into the general-purpose register *r10*.

LOADING CONSTANTS

In ARM processor, there is no instruction to move a 32-bit constant into a register. But there are two pseudo instructions to move a 32-bit value into a register as below.

```
Syntax: LDR Rd, =constant
        ADR Rd, label
```

LDR	load constant pseudoinstruction	*Rd* = 32-bit constant
ADR	load address pseudoinstruction	*Rd* = 32-bit relative address

- The first pseudo instruction writes a 32-bit constant to a register using whatever instructions are available. It defaults to a memory read if the constant cannot be encoded using other instructions.
- The second pseudo instruction writes a relative address into a register, which will be encoded using a *pc*-relative expression.

EXAMPLE **8.28** This example shows an LDR instruction loading a 32-bit constant 0xff00ffff into register *r0*.

```
        LDR     r0, [pc, #constant_number-8-(PC)]
        :
constant_number
        DCD     0xff00ffff
```

This example involves a memory access to load the constant, which can be expensive for time-critical routines.

Example 3.29 shows an alternative method to load the same constant into register *r0* by using an MVN instruction.

Table **8**.12 LDR pseudoinstruction conversion.

Pseudoinstruction	Actual instruction
LDR r0, =0xff	MOV r0, #0xff
LDR r0, =0x55555555	LDR r0, [pc, #offset_12]

EXAMPLE 8.29 Loading the constant 0xff00ffff using an MVN.

```
PRE     none...
        MVN     r0, #0x00ff0000
POST    r0 = 0xff00ffff
```

- As can be seen above, there are alternatives to access memory, but they depend upon the constant you are trying to load.
- Compilers and assemblers use clever techniques to avoid loading a constant from memory.
- These tools have algorithms to find the optimal number of instructions required to generate a constant in a register and make extensive use of the barrel shifter.
- If the tools cannot generate the constant by these methods, then it is loaded from memory.
- The LDR pseudo instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a *literal pool*—a data area embedded within the code.
- Table 8.12 shows two pseudo code conversions. The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load.
- Another useful pseudo instruction is the ADR instruction, or *address relative*. This instruction places the address of the given *label* into register *Rd*, using a *pc*-relative add or subtract.
- As you can see, there are alternatives to accessing memory, but they depend upon the constant you are trying to load.
- Compilers and assemblers use clever techniques to avoid loading a constant from memory.
- These tools have algorithms to find the optimal number of instructions required to generate a constant in a register and make extensive use of the barrel shifter.
- If the tools cannot generate the constant by these methods, then it is loaded from memory.
- The LDR pseudo instruction either inserts an MOV or MVN instruction to generate a value (if possible) or generates an LDR instruction with a *pc*-relative address to read the constant from a *literal pool*—a data area embedded within the code.
- Table 8.12 shows two pseudocode conversions. The first conversion produces a simple MOV instruction; the second conversion produces a *pc*-relative load.
- Another useful pseudo instruction is the ADR instruction, or *address relative*. This instruction places the address of the given *label* into register *Rd*, using a *pc*-relative add or subtract.

****

CHAPTER-2 : C COMPILERS AND OPTIMIZATION

- Optimizing code takes time and reduces source code readability. Usually, it's only worth optimizing functions that are frequently executed and important for performance.
- We recommend you use a performance profiling tool, found in most ARM simulators, to find these frequently executed functions.
- Document nonobvious optimizations with source code comments to aid maintainability.
- C compilers have to translate your C function literally into assembler so that it works for all possible inputs.
- In practice, many of the input combinations are not possible or won't occur. Let's start by looking at an example of the problems the compiler faces.
- The memclr function clears N bytes of memory at address data.

```
void memclr(char *data, int N)
{
  for (; N>0; N--)
  {
    *data=0;
    data++;
  }
}
```

- No matter how advanced the compiler, it does not know whether N can be 0 on input or not. Therefore the compiler needs to test for this case explicitly before the first iteration of the loop.
- The compiler doesn't know whether the data array pointer is four-byte aligned or not. If it is four-byte aligned, then the compiler can clear four bytes at a time using an int store rather than a char store.
- Nor does it know whether N is a multiple of four or not. If N is a multiple of four, then the compiler can repeat the loop body four times or store four bytes at a time using an int store.

To keep our examples concrete, we have tested them using the following specific C compilers:

- armcc from ARM Developer Suite version 1.1 (ADS1.1). You can license this compiler, or a later version, directly from ARM.
- arm-elf-gcc version 2.95.2. This is the ARM target for the GNU C compiler, gcc, and is freely available.

We have used armcc from ADS1.1 to generate the example assembler output in this book. The following short script shows you how to invoke armcc on a C file test.c. You can use this to reproduce our examples.

```
armcc -Otime -c -o test.o test.c
fromelf -text/c test.o > test.txt
```

By default armcc has full optimizations turned on (the -02 command line switch). The -0time switch optimizes for execution efficiency rather than space and mainly affects the layout of for and while loops. If you are using the gcc compiler, then the following short script generates a similar assembler output listing:

```
arm-elf-gcc -02 -fomit-frame-pointer -c -o test.o test.c
arm-elf-objdump -d test.o > test.txt
```

Basic C Data Types

ARM supports operations on different data types.

The data types we can load (or store) can be signed and unsigned words, halfwords, or bytes. The extensions for these data types are: -h or -sh for halfwords, -b or -sb for bytes, and no extension for words. The difference between signed and unsigned data types is:

Signed data types can hold both positive and negative values and are therefore lower in range.

Unsigned data types can hold large positive values (including 'Zero') but cannot hold negative values and are therefore wider in range.

- ARM processors have 32-bit registers and 32-bit data processing operations. The ARM architecture is a RISC load/store architecture.

- In other words you must load values from memory into registers before acting on them. There are no arithmetic or logical instructions that manipulate values in memory directly.

Load and store instructions by ARM architecture.

Architecture	Instruction	Action
Pre-ARMv4	LDRB	load an unsigned 8-bit value
	STRB	store a signed or unsigned 8-bit value
	LDR	load a signed or unsigned 32-bit value
	STR	store a signed or unsigned 32-bit value
ARMv4	LDRSB	load a signed 8-bit value
	LDRH	load an unsigned 16-bit value
	LDRSH	load a signed 16-bit value
	STRH	store a signed or unsigned 16-bit value
ARMv5	LDRD	load a signed or unsigned 64-bit value
	STRD	store a signed or unsigned 64-bit value

- The ARMv4 architecture and above support signed 8-bit and 16-bit loads and stores directly, through new instructions
- ARMv5 adds instruction support for 64-bit load and stores. This is available in ARM9E and later cores.
- Therefore ARM C compilers define char to be an unsigned 8-bit value, rather than a signed 8-bit value as is typical in many other compilers.
- Compilers *armcc* and *gcc* use the datatype mappings
- A common example is using a char type variable i as a loop counter, with loop continuation condition $i \geq 0$.
- As i is unsigned for the ARM compilers, the loop will never terminate. Fortunately *armcc* produces a warning in this situation: *unsigned comparison with 0.*

- Compilers also provide an override switch to make char signed. For example, the command line option -fsigned-char will make char signed on *gcc*.
- The command line option -zc will have the same effect with *armcc*.

C compiler datatype mappings.

C Data Type	Implementation
`char`	unsigned 8-bit byte
`short`	signed 16-bit halfword
`int`	signed 32-bit word
`long`	signed 32-bit word
`long long`	signed 64-bit double word

Local Variable Types

- ARMv4-based processors can efficiently load and store 8-, 16-, and 32-bit data. However, most ARM data processing operations are 32-bit only.
- For this reason, you should use a 32-bit datatype, int or long, for local variables wherever possible.
- Avoid using char and short as local variable types, even if you are manipulating an 8- or 16-bit value.

```
int checksum_v1(int *data)
{
  char i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += data[i];
  }
  return sum;
}
```

Consider the compiler output for this function. We've added labels and comments to make the assembly clear.

```
checksum_v1
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v1_loop
        LDR     r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD     r1,r1,#1            ; r1 = i+1
        AND     r1,r1,#0xff         ; i = (char)r1
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v1_loop    ; if (i<64) loop
        MOV     pc,r14              ; return sum
```

Now compare this to the compiler output where instead we declare i as an `unsigned int`.

```
checksum_v2
        MOV     r2,r0               ; r2 = data
        MOV     r0,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v2_loop
        LDR     r3,[r2,r1,LSL #2]   ; r3 = data[i]
        ADD     r1,r1,#1            ; r1++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r0,r3,r0            ; sum += r3
        BCC     checksum_v2_loop    ; if (i<64) goto loop
        MOV     pc,r14              ; return sum
```

In the first case, the compiler inserts an extra AND instruction to reduce i to the range 0 to 255 before the comparison with 64. This instruction disappears in the second case.

Next, suppose the data packet contains 16-bit values and we need a 16-bit checksum. It is tempting to write the following C code:

```
short checksum_v3(short *data)
{
  unsigned int i;
  short sum=0;

  for (i=0; i<64; i++)
  {
    sum = (short)(sum + data[i]);

          }
          return sum;
        }
```

The loop is now three instructions longer than the loop for example checksum_v2 earlier! There are two reasons for the extra instructions:

- The LDRH instruction does not allow for a shifted address offset as the LDR instruction did in checksum_v2. Therefore the first ADD in the loop calculates the address of item i in the array. The LDRH loads from an address with no offset. LDRH has fewer addressing modes than LDR as it was a later addition to the ARM instruction set.
- The cast reducing total +array[i] to a short requires two MOV instructions. The compiler shifts left by 16 and then right by 16 to implement a 16-bit sign extend. The shift right is a sign-extending shift so it replicates the sign bit to fill the upper 16 bits.

EXAMPLE 5.1

The checksum_v4 code fixes all the problems we have discussed in this section. It uses int type local variables to avoid unnecessary casts. It increments the pointer data instead of using an index offset data[i].

```
short checksum_v4(short *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
     sum += *(data++);
  }
  return (short)sum;
}
```

The *(data++) operation translates to a single ARM instruction that loads the data and increments the data pointer. Of course you could write sum += *data; data++; or even *data++ instead if you prefer. The compiler produces the following output. Three instructions have been removed from the inside loop, saving three cycles per loop compared to checksum_v3.

```
checksum_v4
        MOV     r2,#0               ; sum = 0
        MOV     r1,#0               ; i = 0
checksum_v4_loop
        LDRSH   r3,[r0],#2          ; r3 = *(data++)
        ADD     r1,r1,#1            ; i++
        CMP     r1,#0x40            ; compare i, 64
        ADD     r2,r3,r2            ; sum += r3
        BCC     checksum_v4_loop    ; if (sum<64) goto loop
        MOV     r0,r2,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)sum
        MOV     pc,r14              ; return r0
```

FUNCTION ARGUMENT TYPES

Consider the following simple function, which adds two 16-bit values, halving the second, and returns a 16-bit sum:

```
short add_v1(short a, short b)
{
  return a + (b>>1);
}
```

- The input values a, b, and the return value will be passed in 32-bit ARM registers. Should the compiler assume that these 32-bit values are in the range of a short type, that is, −32,768 to +32,767?
- Or should the compiler force values to be in this range by sign-extending the lowest 16 bits to fill the 32-bit register?
- The compiler must make compatible decisions for the function caller and callee. Either the caller or callee must perform the cast to a short type.
- If the compiler passes arguments wide, then the callee must reduce function arguments to the correct range. If the compiler passes arguments narrow, then the caller must reduce the range.
- If the compiler returns values wide, then the caller must reduce the return value to the correct range. If the compiler returns values narrow, then the callee must reduce the range before returning the value.

For *armcc* in ADS, function arguments are passed narrow and values returned narrow. In other words, the caller casts argument values and the callee casts return values. The compiler uses the ANSI prototype of the function to determine the datatypes of the function arguments.

The *armcc* output for add_v1 shows that the compiler casts the return value to a short type, but does not cast the input values. It assumes that the caller has already ensured that the 32-bit values `r0` and `r1` are in the range of the short type. This shows narrow passing of arguments and return value.

```
add_v1
        ADD     r0,r0,r1,ASR #1     ; r0 = (int)a + ((int)b>>1)
        MOV     r0,r0,LSL #16
        MOV     r0,r0,ASR #16       ; r0 = (short)r0
        MOV     pc,r14              ; return r0
```

The *gcc* compiler we used is more cautious and makes no assumptions about the range of argument value. This version of the compiler reduces the input arguments to the range of a short in both the caller and the callee. It also casts the return value to a short type. Here is the compiled code for `add_v1`:

```
add_v1_gcc
        MOV     r0, r0, LSL #16
        MOV     r1, r1, LSL #16
        MOV     r1, r1, ASR #17         ; r1 = (int)b>>1
        ADD     r1, r1, r0, ASR #16     ; r1 += (int)a
        MOV     r1, r1, LSL #16
        MOV     r0, r1, ASR #16         ; r0 = (short)r1
        MOV     pc, lr                  ; return r0
```

SIGNED VERSUS UNSIGNED TYPES

If your code uses addition, subtraction, and multiplication, then there is no performance difference between signed and unsigned operations. However, there is a difference when it comes to division. Consider the following short example that averages two integers:

```
int average_v1(int a, int b)
{
  return (a+b)/2;
}
```

This compiles to

```
average_v1
        ADD     r0,r0,r1            ; r0 = a + b
        ADD     r0,r0,r0,LSR #31    ; if (r0<0) r0++
        MOV     r0,r0,ASR #1        ; r0 = r0>>1
        MOV     pc,r14              ; return r0
```

Notice that the compiler adds one to the sum before shifting by right if the sum is negative. In other words it replaces $x/2$ by the statement:

```
(x<0) ? ((x+1)>>1): (x>>1)
```

It must do this because x is signed. In C on an ARM target, a divide by two is not a right shift if x is negative. For example, $-3 \gg 1 = -2$ but $-3/2 = -1$. Division rounds towards zero, but arithmetic right shift rounds towards $-\infty$.

It is more efficient to use unsigned types for divisions. The compiler converts unsigned power of two divisions directly to right shifts. For general divisions, the divide routine in the C library is faster for unsigned types.

SUMMARY: The Efficient Use of C Types

- For local variables held in registers, don't use a `char` or `short` type unless 8-bit or 16-bit modular arithmetic is necessary. Use the `signed` or `unsigned int` types instead. Unsigned types are faster when you use divisions.

- For array entries and global variables held in main memory, use the type with the smallest size possible to hold the required data. This saves memory footprint. The ARMv4 architecture is efficient at loading and storing all data widths provided you traverse arrays by incrementing the array pointer. Avoid using offsets from the base of the array with `short` type arrays, as LDRH does not support this.

- Use explicit casts when reading array entries or global variables into local variables, or writing local variables out to array entries. The casts make it clear that for fast operation you are taking a narrow width type stored in memory and expanding it to a wider type in the registers. Switch on *implicit narrowing cast* warnings in the compiler to detect implicit casts.

- Avoid implicit or explicit narrowing casts in expressions because they usually cost extra cycles. Casts on loads or stores are usually free because the load or store instruction performs the cast for you.

- Avoid char and short types for function arguments or return values. Instead use the `int` type even if the range of the parameter is smaller. This prevents the compiler performing unnecessary casts.

C LOOPING STRUCTURES

LOOPS WITH A FIXED NUMBER OF ITERATIONS

Below code shows how the compiler treats a loop with incrementing count `i++`.

```
int checksum_v5(int *data)
{
  unsigned int i;
  int sum=0;

  for (i=0; i<64; i++)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v5
        MOV     r2,r0             ; r2 = data
        MOV     r0,#0             ; sum = 0
        MOV     r1,#0             ; i = 0
checksum_v5_loop
        LDR     r3,[r2],#4        ; r3 = *(data++)
        ADD     r1,r1,#1          ; i++
        CMP     r1,#0x40          ; compare i, 64
        ADD     r0,r3,r0          ; sum += r3
        BCC     checksum_v5_loop  ; if (i<64) goto loop
        MOV     pc,r14            ; return sum
```

It takes three instructions to implement the `for` loop structure:

- An ADD to increment i
- A compare to check if i is less than 64
- A conditional branch to continue the loop if i < 64

This is not efficient. On the ARM, a loop should only use two instructions:

■ A subtract to decrement the loop counter, which also sets the condition code flags on the result

■ A conditional branch instruction

The key point is that the loop counter should count down to zero rather than counting up to some arbitrary limit. Then the comparison with zero is free since the result is stored in the condition flags. Since we are no longer using i as an array index, there is no problem in counting down rather than up.

- For an unsigned loop counter i we can use either of the loop continuation conditions `i!=0` or `i>0`.
- As i can't be negative, they are the same condition. For a signed loop counter, it is tempting to use the condition `i>0` to continue the loop.
- You might expect the compiler to generate the following two instructions to implement the loop:

```
SUBS   r1,r1,#1   ; compare i with 1, i=i-1
BGT    loop       ; if (i+1>1) goto loop
```

LOOPS USING A VARIABLE NUMBER OF ITERATIONS

Now suppose we want our checksum routine to handle packets of arbitrary size. We pass in a variable `N` giving the number of words in the data packet. Using the lessons from the last section we count down until `N` = 0 and don't require an extra loop counter `i`.

The `checksum_v7` example shows how the compiler handles a `for` loop with a variable number of iterations `N`.

```
int checksum_v7(int *data, unsigned int N)
{
  int sum=0;

  for (; N!=0; N--)
  {
    sum += *(data++);
  }
  return sum;
}
```

This compiles to

```
checksum_v7
        MOV     r2,#0             ; sum = 0
        CMP     r1,#0             ; compare N, 0
        BEQ     checksum_v7_end   ; if (N==0) goto end

checksum_v7_loop
        LDR     r3,[r0],#4        ; r3 = *(data++)
        SUBS    r1,r1,#1          ; N-- and set flags
        ADD     r2,r3,r2          ; sum += r3
        BNE     checksum_v7_loop  ; if (N!=0) goto loop
checksum_v7_end
        MOV     r0,r2             ; r0 = sum
        MOV     pc,r14            ; return r0
```

Notice that the compiler checks that `N` is nonzero on entry to the function. Often this check is unnecessary since you know that the array won't be empty. In this case a `do-while` loop gives better performance and code density than a `for` loop.

LOOP UNROLLING

- On ARM7 or ARM9 processors the subtract takes one cycle and the branch three cycles, giving an overhead of four cycles per loop.
- You can save some of these cycles by *unrolling* a loop—repeating the loop body several times, and reducing the number of loop iterations by the same proportion.
- For example, let's unroll our packet checksum example four times.

The following code unrolls our packet checksum loop by four times. We assume that the number of words in the packet `N` is a multiple of four.

```
int checksum_v9(int *data, unsigned int N)
{
  int sum=0;

  do
  {
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    N -= 4;
  } while ( N!=0);
  return sum;
}
```

This compiles to

```
checksum_v9
        MOV r2,#0     ; sum = 0
checksum_v9_loop
        LDR           r3,[r0],#4          ; r3 = *(data++)
        SUBS          r1,r1,#4            ; N -= 4 & set flags
        ADD           r2,r3,r2            ; sum += r3
        LDR           r3,[r0],#4          ; r3 = *(data++)
        ADD           r2,r3,r2            ; sum += r3
        LDR           r3,[r0],#4          ; r3 = *(data++)
        ADD           r2,r3,r2            ; sum += r3
        LDR           r3,[r0],#4          ; r3 = *(data++)
        ADD           r2,r3,r2            ; sum += r3
        BNE           checksum_v9_loop    ; if (N!=0) goto loop
        MOV           r0,r2               ; r0 = sum
        MOV           pc,r14              ; return r0
```

There are two questions you need to ask when unrolling a loop:

- How many times should I unroll the loop?
- What if the number of loop iterations is not a multiple of the unroll amount? For example, what if `N` is not a multiple of four in `checksum_v9`?

To start with the first question, only unroll loops that are important for the overall performance of the application. Otherwise unrolling will increase the code size with little performance benefit. Unrolling may even reduce performance by evicting more important code from the cache.

For the second question, try to arrange it so that array sizes are multiples of your unroll amount. If this isn't possible, then you must add extra code to take care of the leftover cases. This increases the code size a little but keeps the performance high.

This example handles the checksum of any size of data packet using a loop that has been unrolled four times.

```
int checksum_v10(int *data, unsigned int N)
{
  unsigned int i;
  int sum=0;

  for (i=N/4; i!=0; i--)
  {
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
    sum += *(data++);
  }
  for (i=N&3; i!=0; i--)
  {

      sum += *(data++);
    }
    return sum;
  }
```

SUMMARY: Writing Loops Efficiently

- Use loops that count down to zero. Then the compiler does not need to allocate a register to hold the termination value, and the comparison with zero is free.
- Use unsigned loop counters by default and the continuation condition `i!=0` rather than `i>0`. This will ensure that the loop overhead is only two instructions.
- Use `do-while` loops rather than `for` loops when you know the loop will iterate at least once. This saves the compiler checking to see if the loop count is zero.
- Unroll important loops to reduce the loop overhead. Do not overunroll. If the loop overhead is small as a proportion of the total, then unrolling will increase code size and hurt the performance of the cache.
- Try to arrange that the number of elements in arrays are multiples of four or eight. You can then unroll loops easily by two, four, or eight times without worry-

REGISTER ALLOCATION

- The compiler attempts to allocate a processor register to each local variable you use in a C function.
- It will try to use the same register for different local variables if the use of the variables do not overlap.
- When there are more local variables than available registers, the compiler stores the excess variables on the processor stack.
- These variables are called *spilled* or *swapped out* variables since they are written out to memory (in a similar way virtual memory is swapped out to disk).
- Spilled variables are slow to access compared to variables allocated to registers.

To implement a function efficiently, you need to

- minimize the number of spilled variables
- ensure that the most important and frequently accessed variables are stored in registers

First let's look at the number of processor registers the ARM C compilers have available for allocating variables. Below table shows the standard register names and usage when following the ARM-Thumb procedure call standard (ATPCS), which is used in code generated by C compilers.

C compiler register usage.

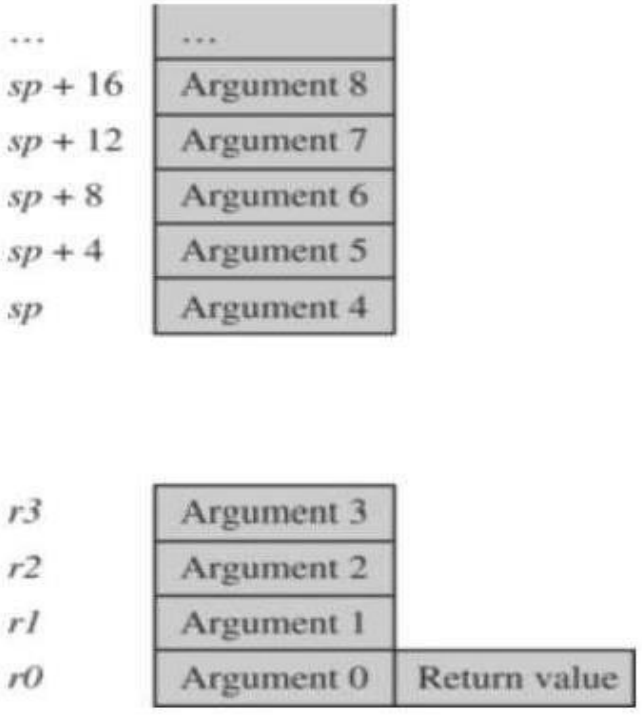
Register number	Alternate register names	ATPCS register usage
r0	*a1*	Argument registers. These hold the first four function arguments on a function call and the return value on a function return. A function may corrupt these registers and use them as general scratch registers within the function.
r1	*a2*	
r2	*a3*	
r3	*a4*	
r4	*v1*	General variable registers. The function must preserve the callee values of these registers.
r5	*v2*	
r6	*v3*	
r7	*v4*	
r8	*v5*	
r9	*v6 sb*	General variable register. The function must preserve the callee value of this register except when compiling for *read-write position independence* (RWPI). Then *r9* holds the *static base* address. This is the address of the read-write data.

r10	*v7 sl*	General variable register. The function must preserve the callee value of this register except when compiling with stack limit checking. Then *r10* holds the stack limit address.
r11	*v8 fp*	General variable register. The function must preserve the callee value of this register except when compiling using a frame pointer. Only old versions of *armcc* use a frame pointer.
r12	*ip*	A general scratch register that the function can corrupt. It is useful as a scratch register for function veneers or other intraprocedure call requirements.
r13	*sp*	The stack pointer, pointing to the full descending stack.
r14	*lr*	The link register. On a function call this holds the return address.
r15	*pc*	The program counter.

- The C compiler can assign 14 variables to registers without spillage.
- In practice, some compilers use a fixed register such as *r12* for intermediate scratch working and do not assign variables to this register.
- Also, complex expressions require intermediate working registers to evaluate. Therefore, to ensure good assignment to registers, you should try to limit the internal loop of functions to using at most 12 local variables.
- If the compiler does need to swap out variables, then it chooses which variables to swap out based on frequency of use.
- A variable used inside a loop counts multiple times. You can guide the compiler as to which variables are important by ensuring these variables are used within the innermost loop.
- The `register` keyword in C hints that a compiler should allocate the given variable to a register.
- However, different compilers treat this keyword in different ways, and different architectures have a different number of available registers (for example, Thumb and ARM).
- Therefore we recommend that you avoid using `register` and rely on the compiler's normal register allocation routine.

FUNCTION CALLS

- The ARM Procedure Call Standard (APCS) defines how to pass function arguments and return values in ARM registers.
- The more recent ARM-Thumb Procedure Call Standard (ATPCS) covers ARM and Thumb interworking as well.
- The first four integer arguments are passed in the first four ARM registers: *r0, r1, r2*, and *r3*. Subsequent integer arguments are placed on the full descending stack, ascending in memory as in figure. Function return integer values are passed in *r0*.



ATPCS argument passing.

- This description covers only integer or pointer arguments. Two-word arguments such as `long long` or `double` are passed in a pair of consecutive argument registers and returned in *r0, r1*.
- The compiler may pass structures in registers or by reference according to command line compiler options.
- The first point to note about the procedure call standard is the *four-register rule*.
- Functions with four or fewer arguments are far more efficient to call than functions with five or more arguments.
- For functions with four or fewer arguments, the compiler can pass all the arguments in registers.
- For functions with more arguments, both the caller and callee must access the stack for some arguments.
- Note that for C++ the first argument to an object method is the *this* pointer. This argument is implicit and additional to the explicit arguments.
- If your C function needs more than four arguments, or your C++ method more than three explicit arguments, then it is almost always more efficient to use structures.
- Group related arguments into structures, and pass a structure pointer rather than multiple arguments. Which arguments are related will depend on the structure of your software.

The next example illustrates the benefits of using a structure pointer. First we show a typical routine to insert `N` bytes from array `data` into a queue. We implement the queue using a cyclic buffer with start address `Q_start` (inclusive) and end address `Q_end` (exclusive).

```
char *queue_bytes_v1(
  char *Q_start,          /* Queue buffer start address */
  char *Q_end,            /* Queue buffer end address */
  char *Q_ptr,            /* Current queue pointer position */
  char *data,             /* Data to insert into the queue */
  unsigned int N)         /* Number of bytes to insert */
{
  do
  {
    *(Q_ptr++) = *(data++);

    if (Q_ptr == Q_end)
    {
      Q_ptr = Q_start;
    }
  } while (--N);
  return Q_ptr;
}
```

This compiles to

```
queue_bytes_v1
        STR     r14,[r13,#-4]!   ; save lr on the stack
        LDR     r12,[r13,#4]     ; r12 = N
queue_v1_loop
        LDRB    r14,[r3],#1      ; r14 = *(data++)
        STRB    r14,[r2],#1      ; *(Q_ptr++) = r14
        CMP     r2,r1            ; if (Q_ptr == Q_end)
        MOVEQ   r2,r0            ;    {Q_ptr = Q_start;}
        SUBS    r12,r12,#1       ; --N and set flags
        BNE     queue_v1_loop    ; if (N!=0) goto loop
        MOV     r0,r2            ; r0 = Q_ptr
        LDR     pc,[r13],#4      ; return r0
```

Compare this with a more structured approach using three function arguments.

Example

The following code creates a `Queue` structure and passes this to the function to reduce the number of function arguments.

```
typedef struct {
  char *Q_start;          /* Queue buffer start address */
  char *Q_end;            /* Queue buffer end address */
  char *Q_ptr;            /* Current queue pointer position */
} Queue;

void queue_bytes_v2(Queue *queue, char *data, unsigned int N)
{
  char *Q_ptr = queue->Q_ptr;
  char *Q_end = queue->Q_end;

  do
  {
    *(Q_ptr++) = *(data++);

    if (Q_ptr == Q_end)
    {
      Q_ptr = queue->Q_start;
    }
  } while (--N);
  queue->Q_ptr = Q_ptr;
}
```

This compiles to

```
queue_bytes_v2
        STR     r14,[r13,#-4]!   ; save lr on the stack
        LDR     r3,[r0,#8]       ; r3 = queue->Q_ptr
        LDR     r14,[r0,#4]      ; r14 = queue->Q_end
queue_v2_loop
        LDRB    r12,[r1],#1      ; r12 = *(data++)
        STRB    r12,[r3],#1      ; *(Q_ptr++) = r12
        CMP     r3,r14           ; if (Q_ptr == Q_end)
        LDREQ   r3,[r0,#0]       ;    Q_ptr = queue->Q_start
        SUBS    r2,r2,#1         ; --N and set flags
        BNE     queue_v2_loop    ; if (N!=0) goto loop
        STR     r3,[r0,#8]       ; queue->Q_ptr = r3
        LDR     pc,[r13],#4      ; return
```

- The `queue_bytes_v2` is one instruction longer than `queue_bytes_v1`, but it is in fact more efficient overall.
- The second version has only three function arguments rather than five. Each call to the function requires only three register setups.
- This compares with four register setups, a stack push, and a stack pull for the first version. There is a net saving of two instructions in function call overhead.
- There are likely further savings in the callee function, as it only needs to assign a single register to the `Queue` structure pointer, rather than three registers in the nonstructured case.

There are other ways of reducing function call overhead if your function is very small and corrupts few registers (uses few local variables). Put the C function in the same C file as the functions that will call it. The C compiler then knows the code generated for the callee function and can make optimizations in the caller function:

■ The caller function need not preserve registers that it can see the callee doesn't corrupt. Therefore the caller function need not save all the ATPCS corruptible registers.

■ If the callee function is very small, then the compiler can inline the code in the caller function. This removes the function call overhead completely.

Example

The function `uint_to_hex` converts a 32-bit unsigned integer into an array of eight hexadecimal digits. It uses a helper function `nybble_to_hex`, which converts a digit d in the range 0 to 15 to a hexadecimal digit.

```
        unsigned int nybble_to_hex(unsigned int d)
        {
          if (d<10)
          {
            return d + '0';

  }
  return d - 10 + 'A';
}

void uint_to_hex(char *out, unsigned int in)
{
  unsigned int i;

  for (i=8; i!=0; i--)
  {
    in = (in<<4) | (in>>28); /* rotate in left by 4 bits */
    *(out++) = (char)nybble_to_hex(in & 15);
  }
}
```

When we compile this, we see that `uint_to_hex` doesn't call `nybble_to_hex` at all! In the following compiled code, the compiler has inlined the `uint_to_hex` code. This is more efficient than generating a function call.

```
uint_to_hex
        MOV     r3,#8               ; i = 8
uint_to_hex_loop
        MOV     r1,r1,ROR #28       ; in = (in<<4)|(in>>28)
        AND     r2,r1,#0xf          ; r2 = in & 15
        CMP     r2,#0xa             ; if (r2>=10)
        ADDCS   r2,r2,#0x37         ;    r2 +='A'-10
        ADDCC   r2,r2,#0x30         ; else r2 +='0'
        STRB    r2,[r0],#1          ; *(out++) = r2
        SUBS    r3,r3,#1            ; i-- and set flags
        BNE     uint_to_hex_loop    ; if (i!=0) goto loop
        MOV     pc,r14              ; return
```

The compiler will only inline small functions. You can ask the compiler to inline a function using the `__inline` keyword, although this keyword is only a hint and the compiler may ignore it. Inlining large functions can lead to big increases in code size without much performance improvement.

SUMMARY: Calling Functions Efficiently

- Try to restrict functions to four arguments. This will make them more efficient to call. Use structures to group related arguments and pass structure pointers instead of multiple arguments.
- Define small functions in the same source file and before the functions that call them. The compiler can then optimize the function call or inline the small function.
- Critical functions can be inlined using the `__inline` keyword.

POINTER ALIASING

- Two pointers are said to *alias* when they point to the same address.
- If you write to one pointer, it will affect the value you read from the other pointer. In a function, the compiler often doesn't know which pointers can alias and which pointers can't.
- The compiler must be very pessimistic and assume that any write to a pointer may affect the value read from any other pointer, which can significantly reduce code efficiency.

Let's start with a very simple example. The following function increments two timer values by a step amount:

```
void timers_v1(int *timer1, int *timer2, int *step)
{
  *timer1 += *step;
  *timer2 += *step;
}
```

This compiles to

```
timers_v1
        LDR     r3,[r0,#0]          ; r3 = *timer1
        LDR     r12,[r2,#0]         ; r12 = *step
        ADD     r3,r3,r12           ; r3 += r12
        STR     r3,[r0,#0]          ; *timer1 = r3
        LDR     r0,[r1,#0]          ; r0 = *timer2
        LDR     r2,[r2,#0]          ; r2 = *step
        ADD     r0,r0,r2            ; r0 += r2
        STR     r0,[r1,#0]          ; *timer2 = t0
        MOV     pc,r14              ; return
```

- Note that the compiler loads from `step` twice. Usually a compiler optimization called *common subexpression elimination* would kick in so that `*step` was only evaluated once, and the value reused for the second occurrence.
- However, the compiler can't use this optimization here. The pointers `timer1` and `step` might alias one another.
- In other words, the compiler cannot be sure that the write to `timer1` doesn't affect the read from step.
- In this case the second value of `*step` is different from the first and has the value `*timer1`. This forces the compiler to insert an extra load instruction.

The same problem occurs if you use structure accesses rather than direct pointer access. The following code also compiles inefficiently:

```
typedef struct {int step;} State;
typedef struct {int timer1, timer2;} Timers;

void timers_v2(State *state, Timers *timers)
{
  timers->timer1 += state->step;
  timers->timer2 += state->step;
}
```

The compiler evaluates `state->step` twice in case `state->step` and timers->timer1 are at the same memory address. The fix is easy: Create a new local variable to hold the value of `state->step` so the compiler only performs a single load.

Example

In the code for `timers_v3` we use a local variable step to hold the value of `state->step`. Now the compiler does not need to worry that `state` may alias with `timers`.

```
void timers_v3(State *state, Timers *timers)
{
     int step = state->step;

     timers->timer1 += step;
     timers->timer2 += step;
}
```

Consider the following example, which reads and then checksums a data packet:

```
int checksum_next_packet(void)
{
  int *data;
  int N, sum=0;

  data = get_next_packet(&N);

  do
  {
    sum += *(data++);
  } while (--N);

  return sum;
}
```

Here `get_next_packet` is a function returning the address and size of the next data packet. The previous code compiles to

```
checksum_next_packet
        STMFD   r13!,{r4,r14}       ; save r4, lr on the stack
        SUB     r13,r13,#8          ; create two stacked variables
        ADD     r0,r13,#4           ; r0 = &N, N stacked
        MOV     r4,#0               ; sum = 0
        BL      get_next_packet     ; r0 = data
checksum_loop
        LDR     r1,[r0],#4          ; r1 = *(data++)
        ADD     r4,r1,r4            ; sum += r1
        LDR     r1,[r13,#4]         ; r1 = N (read from stack)
        SUBS    r1,r1,#1            ; r1-- & set flags
        STR     r1,[r13,#4]         ; N = r1 (write to stack)
        BNE     checksum_loop       ; if (N!=0) goto loop
        MOV     r0,r4               ; r0 = sum
        ADD     r13,r13,#8          ; delete stacked variables
        LDMFD   r13!,{r4,pc}        ; return r0
```

SUMMARY: Avoiding Pointer Aliasing

- Do not rely on the compiler to eliminate common subexpressions involving memory accesses. Instead create new local variables to hold the expression. This ensures the expression is evaluated only once.

- Avoid taking the address of local variables. The variable may be inefficient to access from then on.

STRUCTURE ARRANGEMENT

- The way you lay out a frequently used structure can have a significant impact on its performance and code density.
- There are two issues concerning structures on the ARM: alignment of the structure entries and the overall size of the structure.
- For architectures up to and including ARMv5TE, load and store instructions are only guaranteed to load and store values with address aligned to the size of the access width. Table 5.4 summarizes these restrictions.

Table 5.4

Load and store alignment restrictions for ARMv5TE.

Transfer size	Instruction	Byte address
1 byte	`LDRB, LDRSB, STRB`	any byte address alignment
2 bytes	`LDRH, LDRSH, STRH`	multiple of 2 bytes
4 bytes	`LDR, STR`	multiple of 4 bytes
8 bytes	`LDRD, STRD`	multiple of 8 bytes

For example, consider the structure

```
struct {
  char a;
  int b;
  char c;
  short d;
}
```

For a little-endian memory system the compiler will lay this out adding padding to ensure that the next object is aligned to the size of that object:

Address	+3	+2	+1	+0
+0	pad	pad	pad	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]
+8	d[15,8]	d[7,0]	pad	c

To improve the memory usage, you should reorder the elements

```
struct {
  char a;
  char c;
  short d;
  int b;
}
```

This reduces the structure size from 12 bytes to 8 bytes, with the following new layout:

Address	+3	+2	+1	+0
+0	d[15,8]	d[7,0]	c	a
+4	b[31,24]	b[23,16]	b[15,8]	b[7,0]

Therefore, it is a good idea to group structure elements of the same size, so that the structure layout doesn't contain unnecessary padding. The *armcc* compiler does include a keyword `__packed` that removes all padding. For example, the structure

```
__packed struct {
  char a;
  int b;
  char c;
  short d;
}
```

will be laid out in memory as

Address	+3	+2	+1	+0
+0	b[23,16]	b[15,8]	b[7,0]	a
+4	d[15,8]	d[7,0]	c	b[31,24]

- However, packed structures are slow and inefficient to access. The compiler emulates unaligned load and store operations by using several aligned accesses with data operations to merge the results.
- Only use the__packed keyword where space is far more important than speed and you can't reduce padding by rearragement. Also use it for porting code that assumes a certain structure layout in memory.
- The exact layout of a structure in memory may depend on the compiler vendor and compiler version you use.
- In API (Application Programmer Interface) definitions it is often a good idea to insert any padding that you cannot get rid of into the structure manually.
- This way the structure layout is not ambiguous. It is easier to link code between compiler versions and compiler vendors if you stick to unambiguous structures.

Another point of ambiguity is enum. Different compilers use different sizes for an enumerated type, depending on the range of the enumeration. For example, consider the type

```
typedef enum {
  FALSE,
  TRUE
} Bool;
```

- The armcc in ADS1.1 will treat Bool as a one-byte type as it only uses the values 0 and 1. Bool will only take up 8 bits of space in a structure.
- However, gcc will treat Bool as a word and take up 32 bits of space in a structure. To avoid ambiguity it is best to avoid using enum types in structures used in the API to your code.
- Another consideration is the size of the structure and the offsets of elements within the structure. This problem is most acute when you are compiling for the Thumb instruction set.
- Thumb instructions are only 16 bits wide and so only allow for small element offsets from a structure base pointer.
- Table 5.5 shows the load and store base register offsets available in Thumb.

Table 5.5

Thumb load and store offsets.

Instructions	Offset available from the base register
LDRB, LDRSB, STRB	0 to 31 bytes
LDRH, LDRSH, STRH	0 to 31 halfwords (0 to 62 bytes)
LDR, STR	0 to 31 words (0 to 124 bytes)

Therefore the compiler can only access an 8-bit structure element with a single instruction if it appears within the first 32 bytes of the structure. Similarly, single instructions can only access 16-bit values in the first 64 bytes and 32-bit values in the first 128 bytes. Once you exceed these limits, structure accesses become inefficient.

The following rules generate a structure with the elements packed for maximum efficiency:

- Place all 8-bit elements at the start of the structure.
- Place all 16-bit elements next, then 32-bit, then 64-bit.
- Place all arrays and larger elements at the end of the structure.
- If the structure is too big for a single instruction to access all the elements, then group the elements into substructures. The compiler can maintain pointers to the individual substructures.

PORTABILITY ISSUES

Here is a summary of the issues you may encounter when porting C code to the ARM.

- **The char type**. On the ARM, char is unsigned rather than signed as for many other processors. A common problem concerns loops that use a char loop counter i and the continuation condition $i \geq 0$, they become infinite loops. In this situation, armcc produces a warning of unsigned comparison with zero. You should either use a compiler option to make char signed or change loop counters to type int.
- The **int type**. Some older architectures use a 16-bit int, which may cause problems when moving to ARM's 32-bit int type although this is rare nowadays. Note that expressions are promoted to an int type before evaluation. Therefore if i = -0x1000, the expression i == 0xF000 is true on a 16-bit machine but false on a 32- bit machine.
- **Unaligned data pointers**. Some processors support the loading of short and int typed values from unaligned addresses. A C program may manipulate pointers directly so that they become unaligned, for example, by casting a char * to an int *. ARM architectures up to ARMv5TE do not support unaligned pointers. To detect them, run the program on an ARM with an alignment checking trap. For example, you can configure the ARM720T to data abort on an unaligned access.
- **Endian assumptions.** C code may make assumptions about the endianness of a memory system, for example, by casting a char * to an int *. If you configure the ARM for the same endianness the code is expecting, then there is no issue. Otherwise, you must remove endian-dependent code sequences and replace them by endian-independent ones.
- **Function prototyping.** The armcc compiler passes arguments narrow, that is, reduced to the range of the argument type. If functions are not prototyped correctly, then the function may return the wrong answer. Other compilers that pass arguments wide may give the correct answer even if the function prototype is incorrect. Always use ANSI prototypes.
- **Use of bit-fields.** The layout of bits within a bit-field is implementation and endian dependent. If C code assumes that bits are laid out in a certain order, then the code is not portable.
- **Use of enumerations.** Although enum is portable, different compilers allocate different numbers of bytes to an enum. The gcc compiler will always allocate four bytes to an enum type. The armcc compiler will only allocate one byte if the enum takes only eight-

bit values. Therefore you can't cross-link code and libraries between different compilers if you use enums in an API structure.

- **Inline assembly.** Using inline assembly in C code reduces portability between architectures. You should separate any inline assembly into small inlined functions that can easily be replaced.
- **The volatile keyword**. Use the volatile keyword on the type definitions of ARM memory-mapped peripheral locations. This keyword prevents the compiler from optimizing away the memory access. It also ensures that the compiler generates a data access of the correct type. For example, if you define a memory location as a volatile short type, then the compiler will access it using 16-bit load and store instructions LDRSH and STRH.

CHAPTER 9

Exception and Interrupt Handling

At the heart of an embedded system lie the exception handlers. They are responsible for handling errors, interrupts, and other events generated by the external system. Efficient handlers can dramatically improve system performance. The process of determining a good handling method can be complicated, challenging, and fun.

In this chapter we will cover the theory and practice of handling exceptions, and specifically the handling of interrupts on the ARM processor. The ARM processor has seven exceptions that can halt the normal sequential execution of instructions: Data Abort, Fast Interrupt Request, Interrupt Request, Prefetch Abort, Software Interrupt, Reset, and Undefined Instruction.

This chapter is divided into three main sections:

- *Exception handling.* Exception handling covers the specific details of how the ARM processor handles exceptions.
- *Interrupts.* ARM defines an interrupt as a special type of exception. This section discusses the use of interrupt requests, as well as introducing some of the common terms, features, and mechanisms surrounding interrupt handling.
- *Interrupt handling schemes.* The final section provides a set of interrupt handling methods. Included with each method is an example implementation.

9.1 Exception Handling

An exception is any condition that needs to halt the normal sequential execution of instructions. Examples are when the ARM core is reset, when an instruction fetch or memory access fails, when an undefined instruction is encountered, when a software interrupt instruction is executed, or when an external interrupt has been raised. Exception handling is the method of processing these exceptions.

Most exceptions have an associated software *exception handler*—a software routine that executes when an exception occurs. For instance, a Data Abort exception will have a Data Abort handler. The handler first determines the cause of the exception and then services the exception. Servicing takes place either within the handler or by branching to a specific service routine. The Reset exception is a special case since it is used to initialize an embedded system.

This section covers the following exception handling topics:

- ARM processor mode and exceptions
- Vector table
- Exception priorities
- Link register offsets

9.1.1 ARM Processor Exceptions and Modes

Table 9.1 lists the ARM processor exceptions. Each exception causes the core to enter a specific mode. In addition, any of the ARM processor modes can be entered manually by changing the *cpsr*. *User* and *system* mode are the only two modes that are not entered by a corresponding exception, in other words, to enter these modes you must modify the *cpsr*.

When an exception causes a mode change, the core automatically

- saves the *cpsr* to the *spsr* of the exception mode
- saves the *pc* to the *lr* of the exception mode

Table 9.1 ARM processor exceptions and associated modes.

Exception	Mode	Main purpose
Fast Interrupt Request	*FIQ*	fast interrupt request handling
Interrupt Request	*IRQ*	interrupt request handling
SWI and Reset	*SVC*	protected mode for operating systems
Prefetch Abort and Data Abort	*abort*	virtual memory and/or memory protection handling
Undefined Instruction	*undefined*	software emulation of hardware coprocessors

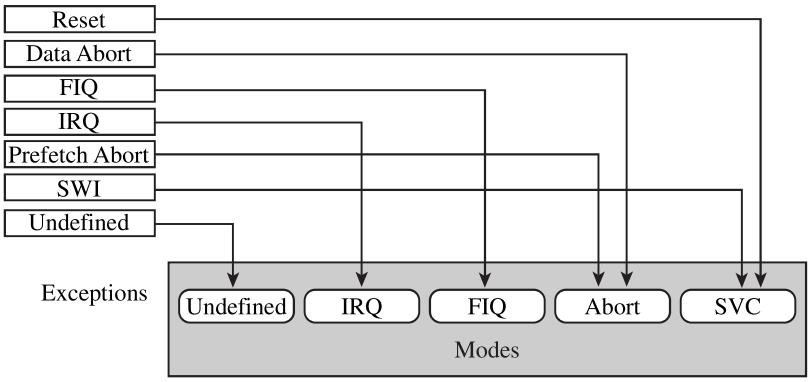


Figure 9.1 Exceptions and associated modes.

- sets the *cpsr* to the exception mode
- sets *pc* to the address of the exception handler

Figure 9.1 shows a simplified view of exceptions and associated modes. Note that when an exception occurs the ARM processor always switches to ARM state.

9.1.2 Vector Table

Chapter 2 introduced the *vector table*—a table of addresses that the ARM core branches to when an exception is raised. These addresses commonly contain branch instructions of one of the following forms:

- `B <address>`—This *branch instruction* provides a branch relative from the *pc*.
- `LDR pc, [pc, #offset]`—This *load register instruction* loads the handler address from memory to the *pc*. The address is an absolute 32-bit value stored close to the vector table. Loading this absolute literal value results in a slight delay in branching to a specific handler due to the extra memory access. However, you can branch to any address in memory.
- `LDR pc, [pc, #-0xff0]`—This *load register instruction* loads a specific interrupt service routine address from address `0xffff030` to the *pc*. This specific instruction is only used when a vector interrupt controller is present (VIC PL190).

Table 9.2 Vector table and processor modes.

Exception	Mode	Vector table offset
Reset	SVC	+0x00
Undefined Instruction	UND	+0x04
Software Interrupt (SWI)	SVC	+0x08
Prefetch Abort	ABT	+0x0c
Data Abort	ABT	+0x10
Not assigned	—	+0x14
IRQ	IRQ	+0x18
FIQ	FIQ	+0x1c

- `MOV pc, #immediate`—This *move instruction* copies an immediate value into the *pc*. It lets you span the full address space but at limited alignment. The address must be an 8-bit immediate rotated right by an even number of bits.

You can also have other types of instructions in the vector table. For example, the FIQ handler might start at address offset `+0x1c`. Thus, the FIQ handler can start immediately at the FIQ vector location, since it is at the end of the vector table. The branch instructions cause the *pc* to jump to a specific location that can handle the specific exception.

Table 9.2 shows the exception, mode, and vector table offset for each exception.

EXAMPLE 9.1 Figure 9.2 shows a typical vector table. The Undefined Instruction entry is a branch instruction to jump to the undefined handler. The other vectors use an indirect address jump with the `LDR` load to *pc* instruction.

Notice that the FIQ handler also uses the `LDR` load to *pc* instruction and does not take advantage of the fact that the handler can be placed at the FIQ vector entry location.

```
0x00000000: 0xe59ffa38  RESET: > ldr  pc, [pc, #reset]
0x00000004: 0xea000502  UNDEF:   b    undInstr
0x00000008: 0xe59ffa38  SWI  :   ldr  pc, [pc, #swi]
0x0000000c: 0xe59ffa38  PABT :   ldr  pc, [pc, #prefetch]
0x00000010: 0xe59ffa38  DABT :   ldr  pc, [pc, #data]
0x00000014: 0xe59ffa38   -   :   ldr  pc, [pc, #notassigned]
0x00000018: 0xe59ffa38  IRQ  :   ldr  pc, [pc, #irq]
0x0000001c: 0xe59ffa38  FIQ  :   ldr  pc, [pc, #fiq]
```

Figure 9.2 Example vector table.

9.1.3 Exception Priorities

Exceptions can occur simultaneously, so the processor has to adopt a priority mechanism. Table 9.3 shows the various exceptions that occur on the ARM processor and their associated priority level. For instance, the Reset exception is the highest priority and occurs when power is applied to the processor. Thus, when a reset occurs, it takes precedence over all other exceptions. Similarly, when a Data Abort occurs, it takes precedence over all other exceptions apart from a Reset exception. The lowest priority level is shared by two exceptions, the Software Interrupt and Undefined Instruction exceptions. Certain exceptions also disable interrupts by setting the *I* or *F* bits in the *cpsr*, as shown in Table 9.3.

Each exception is dealt with according to the priority level set out in Table 9.3. The following is a summary of the exceptions and how they should be handled, starting with the highest.

The Reset exception is the highest priority exception and is always taken whenever it is signaled. The reset handler initializes the system, including setting up memory and caches. External interrupt sources should be initialized before enabling IRQ or FIQ interrupts to avoid the possibility of spurious interrupts occurring before the appropriate handler has been set up. The reset handler must also set up the stack pointers for all processor modes.

During the first few instructions of the handler, it is assumed that no exceptions or interrupts will occur. The code should be designed to avoid SWIs, undefined instructions, and memory accesses that may abort, that is, the handler is carefully implemented to avoid further triggering of an exception.

Data Abort exceptions occur when the memory controller or MMU indicates that an invalid memory address has been accessed (for example, if there is no physical memory for an address) or when the current code attempts to read or write to memory without the correct access permissions. An FIQ exception can be raised within a Data Abort handler since FIQ exceptions are not disabled. When the FIQ is completely serviced, control is returned back to the Data Abort handler.

A Fast Interrupt Request (FIQ) exception occurs when an external peripheral sets the FIQ pin to *nFIQ*. An FIQ exception is the highest priority interrupt. The core disables

Table 9.3 Exception priority levels.

Exceptions	Priority	*I* bit	*F* bit
Reset	1	1	1
Data Abort	2	1	—
Fast Interrupt Request	3	1	1
Interrupt Request	4	1	—
Prefetch Abort	5	1	—
Software Interrupt	6	1	—
Undefined Instruction	6	1	—

both IRQ and FIQ exceptions on entry into the FIQ handler. Thus, no external source can interrupt the processor unless the IRQ and/or FIQ exceptions are reenabled by software. It is desirable that the FIQ handler (and also the abort, SWI, and IRQ handlers) is carefully designed to service the exception efficiently.

An Interrupt Request (IRQ) exception occurs when an external peripheral sets the IRQ pin to *nIRQ*. An IRQ exception is the second-highest priority interrupt. The IRQ handler will be entered if neither an FIQ exception nor Data Abort exception occurs. On entry to the IRQ handler, the IRQ exceptions are disabled and should remain disabled until the current interrupt source has been cleared.

A Prefetch Abort exception occurs when an attempt to fetch an instruction results in a memory fault. This exception is raised when the instruction is in the execute stage of the pipeline and if none of the higher exceptions have been raised. On entry to the handler, IRQ exceptions will be disabled, but the FIQ exceptions will remain unchanged. If FIQ is enabled and an FIQ exception occurs, it can be taken while servicing the Prefetch Abort.

A Software Interrupt (SWI) exception occurs when the `SWI` instruction is executed and none of the other higher-priority exceptions have been flagged. On entry to the handler, the *cpsr* will be set to *supervisor* mode.

If the system uses nested SWI calls, the link register *r14* and *spsr* must be stored away before branching to the nested SWI to avoid possible corruption of the link register and the *spsr*.

An Undefined Instruction exception occurs when an instruction not in the ARM or Thumb instruction set reaches the execute stage of the pipeline and none of the other exceptions have been flagged. The ARM processor "asks" the coprocessors if they can handle this as a coprocessor instruction. Since coprocessors follow the pipeline, instruction identification can take place in the execute stage of the core. If none of the coprocessors claims the instruction, an Undefined Instruction exception is raised.

Both the `SWI` instruction and Undefined Instruction have the same level of priority, since they cannot occur at the same time (in other words, the instruction being executed cannot both be an `SWI` instruction and an undefined instruction).

9.1.4 Link Register Offsets

When an exception occurs, the link register is set to a specific address based on the current *pc*. For instance, when an IRQ exception is raised, the link register *lr* points to the last executed instruction plus 8. Care has to be taken to make sure the exception handler does not corrupt *lr* because *lr* is used to return from an exception handler. The IRQ exception is taken only after the current instruction is executed, so the return address has to point to the next instruction, or $lr - 4$. Table 9.4 provides a list of useful addresses for the different exceptions.

The next three examples show different methods of returning from an IRQ or FIQ exception handler.

Table 9.4 Useful link-register-based addresses.

Exception	Address	Use
Reset	—	*lr* is not defined on a Reset
Data Abort	*lr* − 8	points to the instruction that caused the Data Abort exception
FIQ	*lr* − 4	return address from the FIQ handler
IRQ	*lr* − 4	return address from the IRQ handler
Prefetch Abort	*lr* − 4	points to the instruction that caused the Prefetch Abort exception
SWI	*lr*	points to the next instruction after the SWI instruction
Undefined Instruction	*lr*	points to the next instruction after the undefined instruction

EXAMPLE 9.2 This example shows that a typical method of returning from an IRQ and FIQ handler is to use a SUBS instruction:

```
handler
        <handler code>
        ...
        SUBS    pc, r14, #4             ; pc=r14-4
```

Because there is an S at the end of the SUB instruction and the *pc* is the destination register, the *cpsr* is automatically restored from the *spsr* register.

EXAMPLE 9.3 This example shows another method that subtracts the offset from the link register *r14* at the beginning of the handler.

```
handler
        SUB     r14, r14, #4             ; r14-=4
        ...
        <handler code>
        ...
        MOVS    pc, r14                  ; return
```

After servicing is complete, return to normal execution occurs by moving the link register *r14* into the *pc* and restoring *cpsr* from the *spsr*.

EXAMPLE 9.4 The final example uses the interrupt stack to store the link register. This method first subtracts an offset from the link register and then stores it onto the interrupt stack.

```
handler
        SUB     r14, r14, #4              ; r14-=4
```

```
STMFD   r13!,{r0-r3, r14}          ; store context
...
<handler code>
...
LDMFD   r13!,{r0-r3, pc}^          ; return
```

To return to normal execution, the `LDM` instruction is used to load the *pc*. The ^ symbol in the instruction forces the *cpsr* to be restored from the *spsr*.

9.2 Interrupts

There are two types of interrupts available on the ARM processor. The first type of interrupt causes an exception raised by an external peripheral—namely, IRQ and FIQ. The second type is a specific instruction that causes an exception—the `SWI` instruction. Both types suspend the normal flow of a program.

In this section we will focus mainly on IRQ and FIQ interrupts. We will cover these topics:

- Assigning interrupts
- Interrupt latency
- IRQ and FIQ exceptions
- Basic interrupt stack design and implementation

9.2.1 Assigning Interrupts

A system designer can decide which hardware peripheral can produce which interrupt request. This decision can be implemented in hardware or software (or both) and depends upon the embedded system being used.

An *interrupt controller* connects multiple external interrupts to one of the two ARM interrupt requests. Sophisticated controllers can be programmed to allow an external interrupt source to cause either an IRQ or FIQ exception.

When it comes to assigning interrupts, system designers have adopted a standard design practice:

- Software Interrupts are normally reserved to call privileged operating system routines. For example, an `SWI` instruction can be used to change a program running in *user* mode to a privileged mode. For an SWI handler example, take a look at Chapter 11.
- Interrupt Requests are normally assigned for general-purpose interrupts. For example, a periodic timer interrupt to force a context switch tends to be an IRQ exception. The IRQ exception has a lower priority and higher interrupt latency (to be discussed in the next section) than the FIQ exception.

- Fast Interrupt Requests are normally reserved for a single interrupt source that requires a fast response time—for example, direct memory access specifically used to move blocks of memory. Thus, in an embedded operating system design, the FIQ exception is used for a specific application, leaving the IRQ exception for more general operating system activities.

9.2.2 Interrupt Latency

Interrupt-driven embedded systems have to fight a battle with *interrupt latency*—the interval of time from an external interrupt request signal being raised to the first fetch of an instruction of a specific interrupt service routine (ISR).

Interrupt latency depends on a combination of hardware and software. System architects must balance the system design to handle multiple simultaneous interrupt sources and minimize interrupt latency. If the interrupts are not handled in a timely manner, then the system will exhibit slow response times.

Software handlers have two main methods to minimize interrupt latency. The first method is to use a *nested interrupt handler*, which allows further interrupts to occur even when currently servicing an existing interrupt (see Figure 9.3). This is achieved by reenabling the interrupts as soon as the interrupt source has been serviced (so it won't generate more interrupts) but before the interrupt handling is complete. Once a nested interrupt has been serviced, then control is relinquished to the original interrupt service routine.

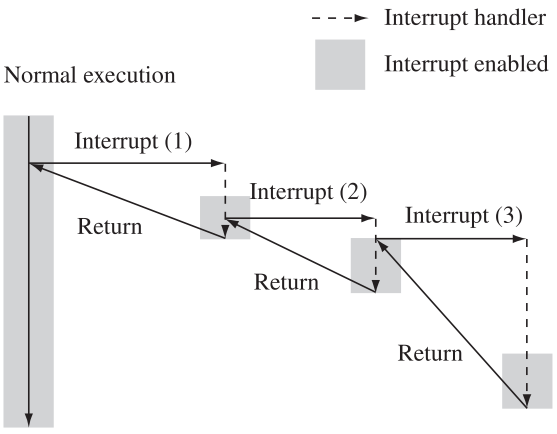


Figure 9.3 A three-level nested interrupt.

The second method involves *prioritization*. You program the interrupt controller to ignore interrupts of the same or lower priority than the interrupt you are handling, so only a higher-priority task can interrupt your handler. You then reenable interrupts.

The processor spends time in the lower-priority interrupts until a higher-priority interrupt occurs. Therefore higher-priority interrupts have a lower average interrupt latency than the lower-priority interrupts, which reduces latency by speeding up the completion time on the critical time-sensitive interrupts.

9.2.3 IRQ AND FIQ EXCEPTIONS

IRQ and FIQ exceptions only occur when a specific interrupt mask is cleared in the *cpsr*. The ARM processor will continue executing the current instruction in the execution stage of the pipeline before handling the interrupt—an important factor in designing a deterministic interrupt handler since some instructions require more cycles to complete the execution stage.

An IRQ or FIQ exception causes the processor hardware to go through a standard procedure (provided the interrupts are not masked):

1. The processor changes to a specific interrupt request mode, which reflects the interrupt being raised.
2. The previous mode's *cpsr* is saved into the *spsr* of the new interrupt request mode.
3. The *pc* is saved in the *lr* of the new interrupt request mode.
4. *Interrupt/s are disabled*—either the IRQ or both IRQ and FIQ exceptions are disabled in the *cpsr*. This immediately stops another interrupt request of the same type being raised.
5. The processor branches to a specific entry in the vector table.

The procedure varies slightly depending upon the type of interrupt being raised. We will illustrate both interrupts with an example. The first example shows what happens when an IRQ exception is raised, and the second example shows what happens when an FIQ exception is raised.

EXAMPLE 9.5 Figure 9.4 shows what happens when an IRQ exception is raised when the processor is in *user* mode. The processor starts in state 1. In this example both the IRQ and FIQ exception bits in the *cpsr* are enabled.

When an IRQ occurs the processor moves into state 2. This transition automatically sets the IRQ bit to one, disabling any further IRQ exceptions. The FIQ exception, however, remains enabled because FIQ has a higher priority and therefore does not get disabled when a low-priority IRQ exception is raised. The *cpsr* processor mode changes to *IRQ* mode. The *user* mode *cpsr* is automatically copied into *spsr_irq*.

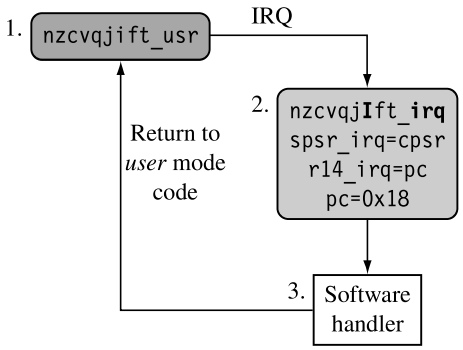


Figure 9.4 Interrupt Request (IRQ).

Register *r14_irq* is assigned the value of the *pc* when the interrupt was raised. The *pc* is then set to the IRQ entry `+0x18` in the vector table.

In state 3 the software handler takes over and calls the appropriate interrupt service routine to service the source of the interrupt. Upon completion, the processor mode reverts back to the original *user* mode code in state 1.

EXAMPLE 9.6 Figure 9.5 shows an example of an FIQ exception. The processor goes through a similar procedure as with the IRQ exception, but instead of just masking further IRQ exceptions from occurring, the processor also masks out further FIQ exceptions. This means that both interrupts are disabled when entering the software handler in state 3.

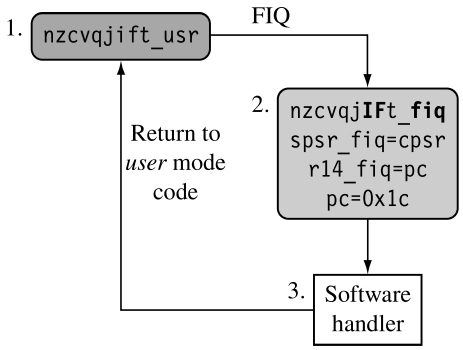


Figure 9.5 Fast Interrupt Request (FIQ).

Changing to *FIQ* mode means there is no requirement to save registers *r8* to *r12* since these registers are banked in *FIQ* mode. These registers can be used to hold temporary data, such as buffer pointers or counters. This makes *FIQ* ideal for servicing a single-source, high-priority, low-latency interrupt.

9.2.3.1 Enabling and Disabling FIQ and IRQ Exceptions

The ARM processor core has a simple procedure to manually enable and disable interrupts that involves modifying the *cpsr* when the processor is in a privileged mode.

Table 9.5 shows how IRQ and FIQ interrupts are enabled. The procedure uses three ARM instructions.

The first instruction `MRS` copies the contents of the *cpsr* into register *r1*. The second instruction clears the IRQ or FIQ mask bit. The third instruction then copies the updated contents in register *r1* back into the *cpsr*, enabling the interrupt request. The postfix `_c` identifies that the bit field being updated is the control field bit [7:0] of the *cpsr*. (For more details see Chapter 2.) Table 9.6 shows a similar procedure to disable or mask an interrupt request.

It is important to understand that the interrupt request is either enabled or disabled only once the `MSR` instruction has completed the execution stage of the pipeline. Interrupts can still be raised or masked prior to the `MSR` completing this stage.

Table 9.5 Enabling an interrupt.

cpsr value	IRQ	FIQ
Pre	*nzcvqj**I**Ft_SVC*	*nzcvqjI**F**t_SVC*

Code	`enable_irq`
	`MRS r1, cpsr`
	`BIC r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`enable_fiq`
		`MRS r1, cpsr`
		`BIC r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqj**i**Ft_SVC*	*nzcvqjI**f**t_SVC*

Table 9.6 Disabling an interrupt.

cpsr	IRQ	**FIQ**
Pre	*nzcvqj**i**ft_SVC*	*nzcvqji**f**t_SVC*

Code	`disable_irq`
	`MRS r1, cpsr`
	`ORR r1, r1, #0x80`
	`MSR cpsr_c, r1`
		`disable_fiq`
		`MRS r1, cpsr`
		`ORR r1, r1, #0x40`
		`MSR cpsr_c, r1`

Post	*nzcvqj**I**ft_SVC*	*nzcvqji**F**t_SVC*

To enable and disable both the IRQ and FIQ exceptions requires a slight modification to the second instruction. The immediate value on the data processing `BIC` or `ORR` instruction has to be changed to `0xc0` to enable or disable both interrupts.

9.2.4 Basic Interrupt Stack Design and Implementation

Exceptions handlers make extensive use of stacks, with each mode having a dedicated register containing the stack pointer. The design of the exception stacks depends upon these factors:

- *Operating system requirements*—Each operating system has its own requirements for stack design.
- *Target hardware*—The target hardware provides a physical limit to the size and positioning of the stack in memory.

Two design decisions need to be made for the stacks:

- The *location* determines where in the memory map the stack begins. Most ARM-based systems are designed with a stack that descends downwards, with the top of the stack at a high memory address.
- *Stack size* depends upon the type of handler, nested or nonnested. A nested interrupt handler requires more memory space since the stack will grow with the number of nested interrupts.

A good stack design tries to avoid *stack overflow*—where the stack extends beyond the allocated memory—because it causes instability in embedded systems. There are software techniques that identify overflow and that allow corrective measures to take place to repair the stack before irreparable memory corruption occurs. The two main methods are (1) to use memory protection and (2) to call a stack check function at the start of each routine.

The *IRQ* mode stack has to be set up before interrupts are enabled—normally in the initialization code for the system. It is important that the maximum size of the stack is known in a simple embedded system, since the stack size is reserved in the initial stages of boot-up by the firmware.

Figure 9.6 shows two typical memory layouts in a linear address space. The first layout, *A*, shows a traditional stack layout with the interrupt stack stored underneath the code segment. The second layout, *B*, shows the interrupt stack at the top of the memory above the user stack. The main advantage of layout *B* over *A* is that *B* does not corrupt the vector table when a stack overflow occurs, and so the system has a chance to correct itself when an overflow has been identified.

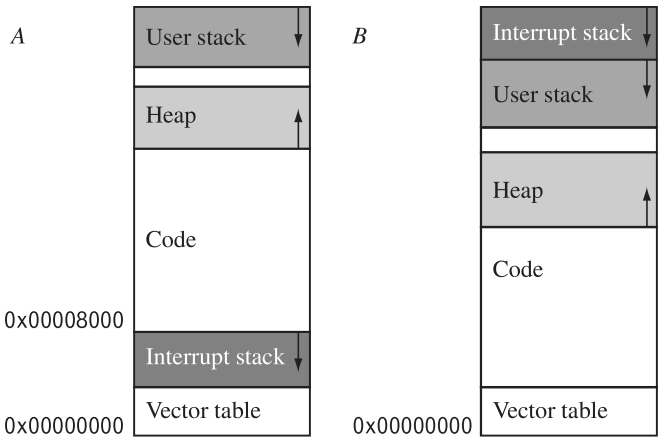


Figure 9.6 Typical memory layouts.

EXAMPLE 9.7 For each processor mode a stack has to be set up. This is carried out every time the processor is reset. Figure 9.7 shows an implementation using layout *A*. To help set up the memory layout, a set of defines are declared that map the memory region names with an absolute address.

For instance, the User stack is given the label `USR_Stack` and is set to address `0x20000`. The Supervisor stack is set to an address that is 128 bytes below the IRQ stack.

```
USR_Stack       EQU 0x20000
IRQ_Stack       EQU 0x8000
SVC_Stack       EQU IRQ_Stack-128
```

To help change to the different processor modes, we declare a set of defines that map each processor mode with a particular mode bit pattern. These labels can then be used to set the *cpsr* to a new mode.

```
Usr32md         EQU 0x10            ; User mode
FIQ32md         EQU 0x11            ; FIQ mode
IRQ32md         EQU 0x12            ; IRQ mode
SVC32md         EQU 0x13            ; Supervisor mode
Abt32md         EQU 0x17            ; Abort mode
Und32md         EQU 0x1b            ; Undefined instruction mode
Sys32md         EQU 0x1f            ; System mode
```

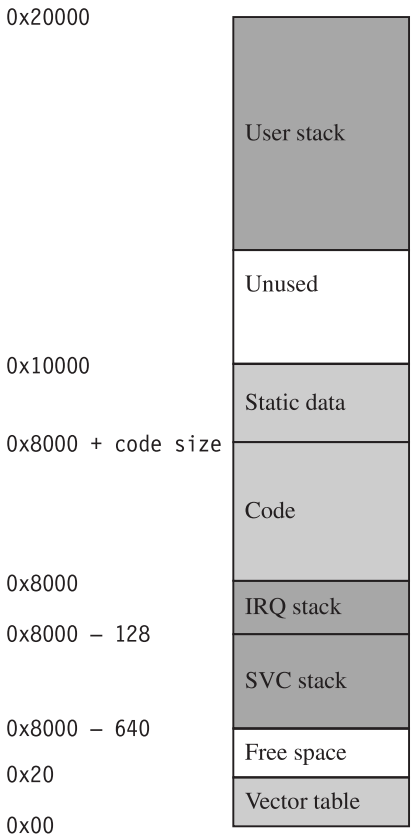


Figure 9.7 Example implementation using layout *A*.

For safety reasons a define is declared to disable both the IRQ and FIQ exceptions in the *cpsr*:

```
NoInt          EQU 0xc0              ; Disable interrupts
```

`NoInt` masks both interrupts by setting the masks to one.

Initialization code starts by setting up the stack registers for each processor mode. The stack register *r13* is one of the registers that is always banked when a mode change occurs. The code first initializes the IRQ stack. For safety reasons, it is always best to make sure that interrupts are disabled by using a bitwise OR between `NoInt` and the new mode.

Each mode stack must be set up. Here is an example of how to set up three different stacks when the processor core comes out of *reset*. Note that, since this is a basic example, we do not implement a stack for the *abort*, *FIQ*, and *undefined instruction* modes. If these stacks are required, then very similar code is used.

- *Supervisor mode stack*—The processor core starts in *supervisor* mode so the SVC stack setup involves loading register *r13_svc* with the address pointed to by `SVC_NewStack`. For this example the value is `SVC_Stack`.

```
        LDR     r13, SVC_NewStack        ; r13_svc
        ...
SVC_NewStack
        DCD   SVC_Stack
```

- *IRQ mode stack*—To set up the IRQ stack, the processor mode has to change to *IRQ* mode. This is achieved by storing a *cpsr* bit pattern into register *r2*. Register *r2* is then copied into the *cpsr*, placing the processor into *IRQ* mode. This action immediately makes register *r13_irq* viewable, and it can then be assigned the `IRQ_Stack` value.

```
        MOV     r2, #NoInt|IRQ32md
        MSR     cpsr_c, r2
        LDR     r13, IRQ_NewStack        ; r13_irq
        ...
IRQ_NewStack
        DCD     IRQ_Stack
```

- *User mode stack*—It is common for the *user* mode stack to be the last to be set up because when the processor is in *user* mode there is no direct method to modify the *cpsr*. An alternative is to force the processor into *system* mode to set up the *user* mode stack since both modes share the same registers.

```
        MOV     r2, #Sys32md
        MSR     cpsr_c, r2
        LDR     r13, USR_NewStack        ; r13_usr
        ...
USR_NewStack
        DCD     USR_Stack
```

Using separate stacks for each mode rather than processing using a single stack has one main advantage: errant tasks can be debugged and isolated from the rest of the system.

9.3 Interrupt Handling Schemes

In this final section we will introduce a number of different interrupt handling schemes, ranging from the simple nonnested interrupt handler to the more complex grouped prioritized interrupt handler. Each scheme is presented as a reference with a general description plus an example implementation.

The schemes covered are the following:

- A *nonnested interrupt handler* handles and services individual interrupts sequentially. It is the simplest interrupt handler.
- A *nested interrupt handler* handles multiple interrupts without a priority assignment.
- A *reentrant interrupt handler* handles multiple interrupts that can be prioritized.
- A *prioritized simple interrupt handler* handles prioritized interrupts.
- A *prioritized standard interrupt handler* handles higher-priority interrupts in a shorter time than lower-priority interrupts.
- A *prioritized direct interrupt handler* handles higher-priority interrupts in a shorter time and goes directly to a specific service routine.
- A *prioritized grouped interrupt handler* is a mechanism for handling interrupts that are grouped into different priority levels.
- A *VIC PL190 based interrupt service routine* shows how the vector interrupt controller (VIC) changes the design of an interrupt service routine.

9.3.1 Nonnested Interrupt Handler

The simplest interrupt handler is a handler that is nonnested: the interrupts are disabled until control is returned back to the interrupted task or process. Because a nonnested interrupt handler can only service a single interrupt at a time, handlers of this form are not suitable for complex embedded systems that service multiple interrupts with differing priority levels.

Figure 9.8 shows the various stages that occur when an interrupt is raised in a system that has implemented a simple nonnested interrupt handler:

1. *Disable interrupt/s*—When the IRQ exception is raised, the ARM processor will disable further IRQ exceptions from occurring. The processor mode is set to the appropriate interrupt request mode, and the previous *cpsr* is copied into the newly available *spsr_{interrupt request mode}*. The processor will then set the *pc* to point to the correct entry in the vector table and execute the instruction. This instruction will alter the *pc* to point to the specific interrupt handler.
2. *Save context*—On entry the handler code saves a subset of the current processor mode nonbanked registers.

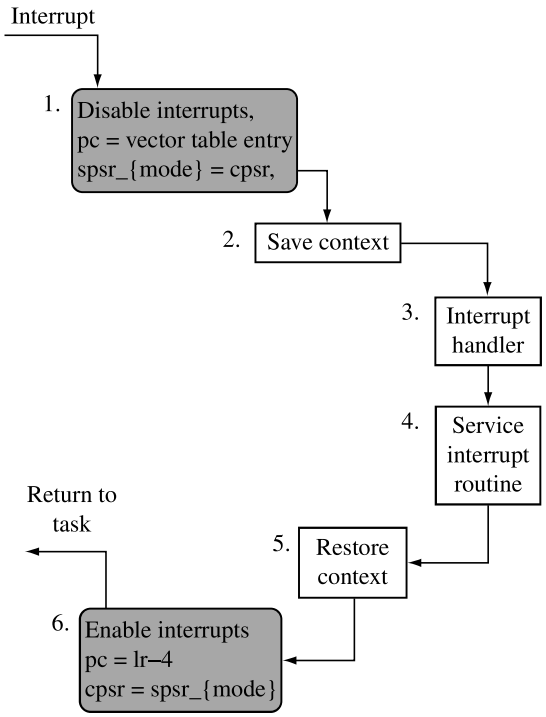


Figure 9.8 Simple nonnested interrupt handler.

3. *Interrupt handler*—The handler then identifies the external interrupt source and executes the appropriate interrupt service routine (ISR).
4. *Interrupt service routine*—The ISR services the external interrupt source and resets the interrupt.
5. *Restore context*—The ISR returns back to the interrupt handler, which restores the context.
6. *Enable interrupts*—Finally, to return from the interrupt handler, the *spsr_{interrupt request mode}* is restored back into the *cpsr*. The *pc* is then set to the next instruction after the interrupt was raised.

EXAMPLE 9.8 This IRQ handler example assumes that the IRQ stack has been correctly set up by the initialization code.

```
interrupt_handler
        SUB     r14,r14,#4                      ; adjust lr
        STMFD   r13!,{r0-r3,r12,r14}            ; save context
        <interrupt service routine>
        LDMFD   r13!,{r0-r3,r12,pc}^            ; return
```

The first instruction sets the link register *r14_irq* to return back to the correct location in the interrupted task or process. As described in Section 9.1.4, due to the pipeline, on entry to an IRQ handler the link register points four bytes beyond the return address, so the handler must subtract four from the link register to account for this discrepancy. The link register is stored on the stack. To return to the interrupted task, the link register contents are restored from the stack and moved into the *pc*.

Notice registers *r0* to *r3* and register *r12* are also preserved because of the ATPCS. This allows an ATPCS-compliant subroutine to be called within the handler.

The `STMFD` instruction saves the context by placing a subset of the registers onto the stack. To reduce interrupt latency we save a minimum number of registers because the time taken to execute an `STMFD` or `LDMFD` instruction is proportional to the number of registers being transferred. The registers are saved to the stack pointed to by the register *r13_{interrupt request mode}*.

If you are using a high-level language within your system it is important to understand the compiler's procedure calling convention because it will influence both the registers saved and the order they are saved onto the stack. For instance, the ARM compilers preserves registers *r4* to *r11* within a subroutine call so there is no need to preserve them unless they will be used by the interrupt handler. If no C routines are called, it may not be necessary to save all of the registers. It is safe to call a C function only when the registers have been saved onto the interrupt stack.

Within a nonnested interrupt handler, it is not necessary to save the *spsr* because it will not be destroyed by any subsequent interrupt.

At the end of the handler the `LDMFD` instruction will restore the context and return from the interrupt handler. The `^` at the end of the `LDMFD` instruction means that the *cpsr* will be restored from the *spsr*, which is only valid if the *pc* is loaded at the same time. If the *pc* is not loaded, then `^` will restore the *user* bank registers.

In this handler all processing is handled within the interrupt handler, which returns directly to the application.

Once the interrupt handler has been entered and the context has been saved, the handler must determine the interrupt source. The following code shows a simple example of how to determine the interrupt source. `IRQStatus` is the address of the interrupt status register. If the interrupt source is not determined, then control can pass to another handler. In this example we pass control to the debug monitor. Alternatively we could just ignore the interrupt.

```
interrupt_handler
        SUB     r14,r14,#4              ; r14-=4
```

```
STMFD   sp!,{r0-r3,r12,r14}    ; save context
LDR     r0,=IRQStatus          ; interrupt status addr
LDR     r0,[r0]                ; get interrupt status
TST     r0,#0x0080             ; if counter timer
BNE     timer_isr              ;   then branch to ISR
TST     r0,#0x0001             ; else if button press
BNE     button_isr             ;   then call button ISR
LDMFD   sp!,{r0-r3,r12,r14}    ; restore context
LDR     pc,=debug_monitor      ; else debug monitor
```

In the preceding code there are two ISRs: `timer_isr` and `button_isr`. They are mapped to specific bits in the `IRQStatus` register, `0x0080` and `0x0001`, respectively.

SUMMARY **Simple Nonnested Interrupt Handler**

- Handles and services individual interrupts sequentially.
- High interrupt latency; cannot handle further interrupts occurring while an interrupt is being serviced.
- Advantages: relatively easy to implement and debug.
- Disadvantage: cannot be used to handle complex embedded systems with multiple priority interrupts.

9.3.2 Nested Interrupt Handler

A nested interrupt handler allows for another interrupt to occur within the currently called handler. This is achieved by reenabling the interrupts before the handler has fully serviced the current interrupt.

For a real-time system this feature increases the complexity of the system but also improves its performance. The additional complexity introduces the possibility of subtle timing issues that can cause a system failure, and these subtle problems can be extremely difficult to resolve. A nested interrupt method is designed carefully so as to avoid these types of problems. This is achieved by protecting the context restoration from interruption, so that the next interrupt will not fill the stack (cause stack overflow) or corrupt any of the registers.

The first goal of any nested interrupt handler is to respond to interrupts quickly so the handler neither waits for asynchronous exceptions, nor forces them to wait for the handler. The second goal is that execution of regular synchronous code is not delayed while servicing the various interrupts.

The increase in complexity means that the designers have to balance efficiency with safety, by using a defensive coding style that assumes problems will occur. The handler has to check the stack and protect against register corruption where possible.

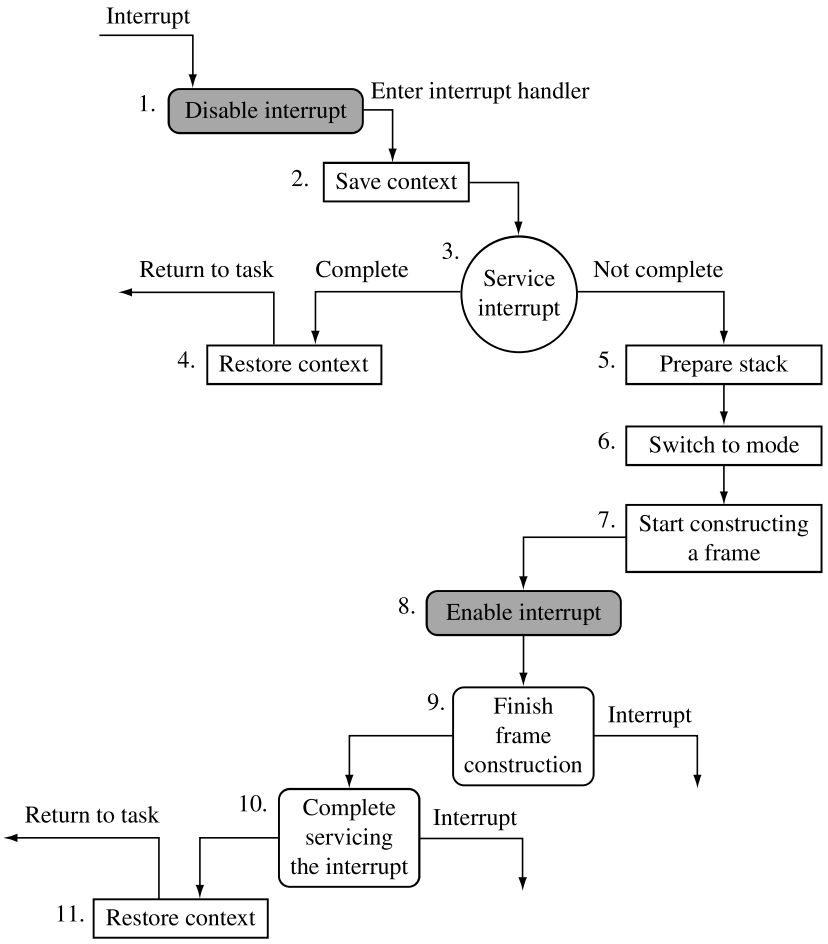


Figure 9.9 Nested interrupt handler.

Figure 9.9 shows a nested interrupt handler. As can been seen from the diagram, the handler is quite a bit more complicated than the simple nonnested interrupt handler described in Section 9.3.1.

The nested interrupt handler entry code is identical to the simple nonnested interrupt handler, except that on exit, the handler tests a flag that is updated by the ISR. The flag indicates whether further processing is required. If further processing is not required, then the interrupt service routine is complete and the handler can exit. If further processing is

required, the handler may take several actions: reenabling interrupts and/or performing a context switch.

Reenabling interrupts involves switching out of *IRQ* mode to either *SVC* or *system* mode. Interrupts cannot simply be reenabled when in *IRQ* mode because this would lead to possible link register *r14_irq* corruption, especially if an interrupt occurred after the execution of a `BL` instruction. This problem will be discussed in more detail in Section 9.3.3.

Performing a context switch involves flattening (emptying) the IRQ stack because the handler does not perform a context switch while there is data on the IRQ stack. All registers saved on the IRQ stack must be transferred to the task's stack, typically on the SVC stack. The remaining registers must then be saved on the task stack. They are transferred to a reserved block of memory on the stack called a *stack frame*.

EXAMPLE 9.9 This nested interrupt handler example is based on the flow diagram in Figure 9.9. The rest of this section will walk through the handler and describe in detail the various stages.

```
Maskmd          EQU 0x1f                ; processor mode mask
SVC32md         EQU 0x13                ; SVC mode
I_Bit           EQU 0x80                ; IRQ bit

FRAME_R0        EQU 0x00
FRAME_R1        EQU FRAME_R0+4
FRAME_R2        EQU FRAME_R1+4
FRAME_R3        EQU FRAME_R2+4
FRAME_R4        EQU FRAME_R3+4
FRAME_R5        EQU FRAME_R4+4
FRAME_R6        EQU FRAME_R5+4
FRAME_R7        EQU FRAME_R6+4
FRAME_R8        EQU FRAME_R7+4
FRAME_R9        EQU FRAME_R8+4
FRAME_R10       EQU FRAME_R9+4
FRAME_R11       EQU FRAME_R10+4
FRAME_R12       EQU FRAME_R11+4
FRAME_PSR       EQU FRAME_R12+4
FRAME_LR        EQU FRAME_PSR+4
FRAME_PC        EQU FRAME_LR+4
FRAME_SIZE      EQU FRAME_PC+4

IRQ_Entry ; instruction                  state : comment
        SUB     r14,r14,#4              ; 2 :
        STMDB   r13!,{r0-r3,r12,r14}    ; 2 : save context
        <service interrupt>
        BL      read_RescheduleFlag     ; 3 : more processing
```

```
CMP     r0,#0                          ; 3 : if processing?
LDMNEIA r13!,{r0-r3,r12,pc}^           ; 4 :   else return
MRS     r2,spsr                        ; 5 : copy spsr_irq
MOV     r0,r13                         ; 5 : copy r13_irq
ADD     r13,r13,#6*4                   ; 5 : reset stack
MRS     r1,cpsr                        ; 6 : copy cpsr
BIC     r1,r1,#Maskmd                  ; 6 :
ORR     r1,r1,#SVC32md                 ; 6 :
MSR     cpsr_c,r1                      ; 6 : change to SVC
SUB     r13,r13,#FRAME_SIZE-FRAME_R4   ; 7 : make space
STMIA   r13,{r4-r11}                   ; 7 : save r4-r11
LDMIA   r0,{r4-r9}                     ; 7 : restore r4-r9
BIC     r1,r1,#I_Bit                   ; 8 :
MSR     cpsr_c,r1                      ; 8 : enable IRA
STMDB   r13!,{r4-r7}                   ; 9 : save r4-r7 SVC
STR     r2,[r13,#FRAME_PSR]            ; 9 : save PSR
STR     r8,[r13,#FRAME_R12]            ; 9 : save r12
STR     r9,[r13,#FRAME_PC]             ; 9 : save pc
STR     r14,[r13,#FRAME_LR]            ; 9 : save lr
<complete interrupt service routine>
LDMIA   r13!,{r0-r12,r14}              ; 11 : restore context
MSR     spsr_cxsf,r14                  ; 11 : restore spsr
LDMIA   r13!,{r14,pc}^                 ; 11 : return
```

This example uses a stack frame structure. All registers are saved onto the frame except for the stack register *r13*. The order of the registers is unimportant except that FRAME_LR and FRAME_PC should be the last two registers in the frame because we will return with a single instruction:

```
LDMIA r13!, {r14, pc}^
```

There may be other registers that are required to be saved onto the stack frame, depending upon the operating system or application being used. For example:

- Registers *r13_usr* and *r14_usr* are saved when there is a requirement by the operating system to support both *user* and *SVC* modes.
- Floating-point registers are saved when the system uses hardware floating point.

There are a number of defines declared in this example. These defines map various *cpsr*/*spsr* changes to a particular label (for example, the *I_Bit*).

A set of defines is also declared that maps the various frame register references with frame pointer offsets. This is useful when the interrupts are reenabled and registers have to be stored into the stack frame. In this example we store the stack frame on the SVC stack.

The entry point for this example handler uses the same code as for the simple nonnested interrupt handler. The link register *r14* is first modified so that it points to the correct return address, and then the context plus the link register *r14* are saved onto the IRQ stack.

An interrupt service routine then services the interrupt. When servicing is complete or partially complete, control is passed back to the handler. The handler then calls a function called `read_RescheduleFlag`, which determines whether further processing is required. It returns a nonzero value in register *r0* if no further processing is required; otherwise it returns a zero. Note we have not included the source for `read_RescheduleFlag` because it is implementation specific.

The return flag in register *r0* is then tested. If the register is not equal to zero, the handler restores context and returns control back to the suspended task.

Register *r0* is set to zero, indicating that further processing is required. The first operation is to save the *spsr*, so a copy of the *spsr_irq* is moved into register *r2*. The *spsr* can then be stored in the stack frame by the handler later on in the code.

The IRQ stack address pointed to by register *r13_irq* is copied into register *r0* for later use. The next step is to flatten (empty) the IRQ stack. This is done by adding 6 * 4 bytes to the top of the stack because the stack grows downwards and an `ADD` instruction can be used to set the stack.

The handler does not need to worry about the data on the IRQ stack being corrupted by another nested interrupt because interrupts are still disabled and the handler will not reenable the interrupts until the data on the IRQ stack has been recovered.

The handler then switches to *SVC* mode; interrupts are still disabled. The *cpsr* is copied into register *r1* and modified to set the processor mode to *SVC*. Register *r1* is then written back into the *cpsr*, and the current mode changes to *SVC* mode. A copy of the new *cpsr* is left in register *r1* for later use.

The next stage is to create a stack frame by extending the stack by the stack frame size. Registers *r4* to *r11* can be saved onto the stack frame, which will free up enough registers to allow us to recover the remaining registers from the IRQ stack still pointed to by register *r0*.

At this stage the stack frame will contain the information shown in Table 9.7. The only registers that are not in the frame are the registers that are stored upon entry to the IRQ handler.

Table 9.8 shows the registers in SVC mode that correspond to the existing IRQ registers. The handler can now retrieve all the data from the IRQ stack, and it is safe to reenable interrupts.

IRQ exceptions are reenabled, and the handler has saved all the important registers. The handler can now complete the stack frame. Table 9.9 shows a completed stack frame that can be used either for a context switch or to handle a nested interrupt.

At this stage the remainder of the interrupt servicing may be handled. A context switch may be performed by saving the current value of register *r13* in the current task's control block and loading a new value for register *r13* from the new task's control block.

It is now possible to return to the interrupted task/handler, or to another task if a context switch occurred.

Table 9.7 SVC stack frame.

Label	Offset	Register
FRAME_R0	+0	—
FRAME_R1	+4	—
FRAME_R2	+8	—
FRAME_R3	+12	—
FRAME_R4	+16	*r4*
FRAME_R5	+20	*r5*
FRAME_R6	+24	*r6*
FRAME_R7	+28	*r7*
FRAME_R8	+32	*r8*
FRAME_R9	+36	*r9*
FRAME_R10	+40	*r10*
FRAME_R11	+44	*r11*
FRAME_R12	+48	—
FRAME_PSR	+52	—
FRAME_LR	+56	—
FRAME_PC	+60	—

Table 9.8 Data retrieved from the IRQ stack.

Registers (SVC)	Retrieved IRQ registers
r4	*r0*
r5	*r1*
r6	*r2*
r7	*r3*
r8	*r12*
r9	*r14* (return address)

SUMMARY **Nested Interrupt Handler**

- Handles multiple interrupts without a priority assignment.
- Medium to high interrupt latency.
- Advantage—can enable interrupts before the servicing of an individual interrupt is complete reducing interrupt latency.
- Disadvantage—does not handle prioritization of interrupts, so lower priority interrupts can block higher priority interrupts.

Table 9.9 Complete frame stack.

Label	Offset	Register
FRAME_R0	+0	*r0*
FRAME_R1	+4	*r1*
FRAME_R2	+8	*r2*
FRAME_R3	+12	*r3*
FRAME_R4	+16	*r4*
FRAME_R5	+20	*r5*
FRAME_R6	+24	*r6*
FRAME_R7	+28	*r7*
FRAME_R8	+32	*r8*
FRAME_R9	+36	*r9*
FRAME_R10	+40	*r10*
FRAME_R11	+44	*r11*
FRAME_R12	+48	*r12*
FRAME_PSR	+52	*spsr_irq*
FRAME_LR	+56	*r14*
FRAME_PC	+60	*r14_irq*

9.3.3 Reentrant Interrupt Handler

A reentrant interrupt handler is a method of handling multiple interrupts where interrupts are filtered by priority, which is important if there is a requirement that interrupts with higher priority have a lower latency. This type of filtering cannot be achieved using the conventional nested interrupt handler.

The basic difference between a reentrant interrupt handler and a nested interrupt handler is that the interrupts are reenabled early on in the reentrant interrupt handler, which can reduce interrupt latency. There are a number of issues relating to reenabling interrupts early, which will be described in more detail later on in this section.

All interrupts in a reentrant interrupt handler must be serviced in *SVC*, *system*, *undefined instruction*, or *abort* mode on the ARM processor.

If interrupts are reenabled in an interrupt mode and the interrupt routine performs a BL subroutine call instruction, the subroutine return address will be set in the register *r14_irq*. This address would be subsequently destroyed by an interrupt, which would overwrite the return address into register *r14_irq*. To avoid this, the interrupt routine should swap into *SVC* or *system* mode. The BL instruction can then use register *r14_svc* to store the subroutine return address. The interrupts must be disabled at the source by setting a bit in the interrupt controller before reenabling interrupts via the *cpsr*.

If interrupts are reenabled in the *cpsr* before processing is complete and the interrupt source is not disabled, an interrupt will be immediately regenerated, leading to an infinite interrupt sequence or *race condition*. Most interrupt controllers have an interrupt mask

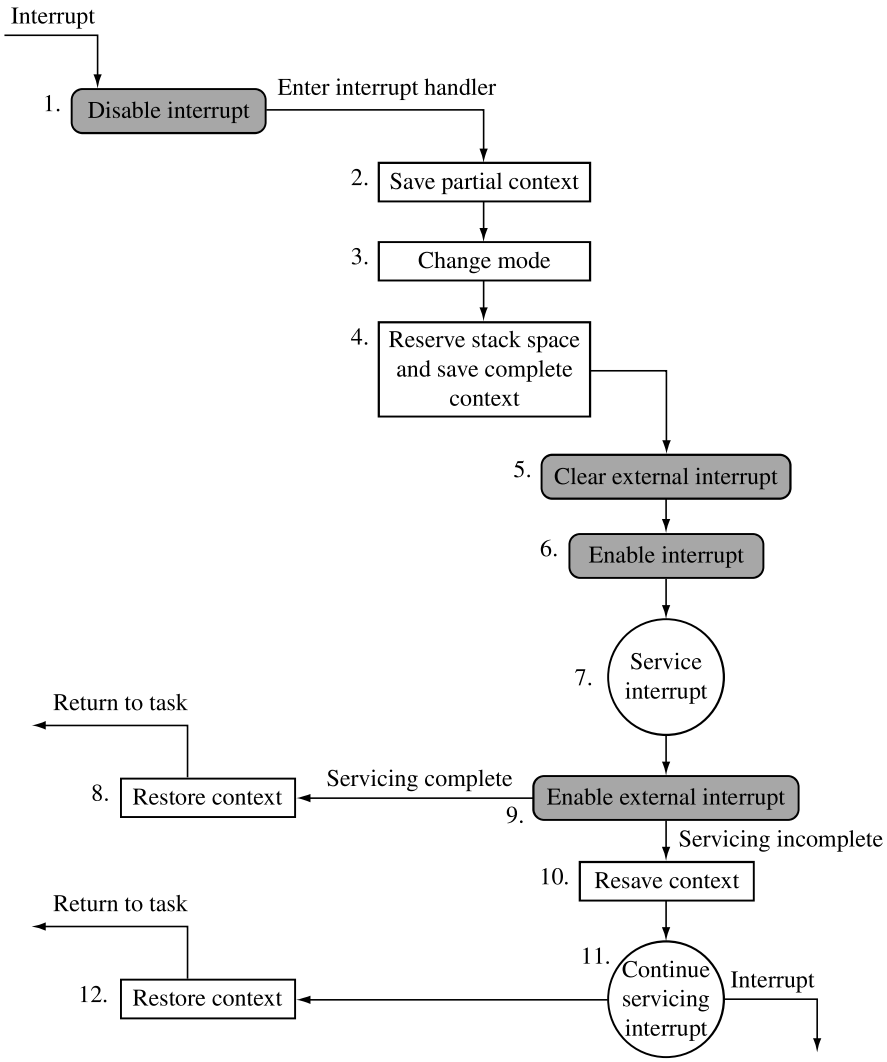


Figure 9.10 Reentrant interrupt handler.

register that allows you to mask out one or more interrupts, but the remaining interrupts are still enabled.

The interrupt stack is unused since interrupts are serviced in *SVC* mode (for example, on the task's stack). Instead the IRQ stack register *r13* is used to point to a 12-byte structure that will be used to store some registers temporarily on interrupt entry.

It is paramount to prioritize interrupts in a reentrant interrupt handler. If the interrupts are not prioritized, the system latency degrades to that of a nested interrupt handler because lower-priority interrupts will be able to preempt the servicing of a higher-priority interrupt. This in turn leads to the locking out of higher-priority interrupts for the duration of the servicing of a lower-priority interrupt.

EXAMPLE 9.10 It is assumed that register *r13_irq* has been set up to point to a 12-byte data structure and does not point to a standard IRQ stack. Offsets such as IRQ_SPSR are used to point into the data structure. As with all interrupt handlers, there are some standard definitions that are required to modify the *cpsr* and *spsr* registers.

```
IRQ_R0          EQU 0
IRQ_spsr        EQU 4
IRQ_R14         EQU 8

Maskmd          EQU 0x1f                ; mask mode
SVC32md         EQU 0x13                ; SVC mode
I_Bit           EQU 0x80                ; IRQ bit

ic_Base         EQU 0x80000000
IRQStatus       EQU 0x0
IRQRawStatus    EQU 0x4
IRQEnable       EQU 0x8
IRQEnableSet    EQU 0x8
IRQEnableClear  EQU 0xc

IRQ_Entry ; instruction                     state : comment
        SUB     r14, r14, #4               ; 2 : r14_irq-=4
        STR     r14, [r13, #IRQ_R14]       ; 2 : save r14_irq
        MRS     r14, spsr                  ; 2 : copy spsr
        STR     r14, [r13, #IRQ_spsr]      ; 2 : save spsr
        STR     r0, [r13, #IRQ_R0]         ; 2 : save r0
        MOV     r0, r13                    ; 2 : copy r13_irq
        MRS     r14, cpsr                  ; 3 : copy cpsr
        BIC     r14, r14, #Maskmd          ; 3 :
        ORR     r14, r14, #SVC32md         ; 3 :
        MSR     cpsr_c, r14                ; 3 : enter SVC mode
        STR     r14, [r13, #-8]!           ; 4 : save r14
        LDR     r14, [r0, #IRQ_R14]        ; 4 : r14_svc=r14_irq
        STR     r14, [r13, #4]             ; 4 : save r14_irq
        LDR     r14, [r0, #IRQ_spsr]       ; 4 : r14_svc=spsr_irq
        LDR     r0, [r0, #IRQ_R0]          ; 4 : restore r0
        STMDB   r13!, {r0-r3,r8,r12,r14}   ; 4 : save context
```

```
LDR     r14, =ic_Base              ; 5 : int crtl address
LDR     r8, [r14, #IRQStatus]      ; 5 : get int status
STR     r8, [r14, #IRQEnableClear] ; 5 : clear interrupts
MRS     r14, cpsr                  ; 6 : r14_svc=cpsr
BIC     r14, r14, #I_Bit           ; 6 : clear I-Bit
MSR     cpsr_c, r14                ; 6 : enable IRQ int
BL      process_interrupt          ; 7 : call ISR
LDR     r14, =ic_Base              ; 9 : int ctrl address
STR     r8, [r14, #IRQEableSet]    ; 9 : enable ints
BL      read_RescheduleFlag        ; 9 : more processing
CMP     r0, #0                     ; 8 : if processing
LDMNEIA r13!, {r0-r3,r8,r12,r14}   ; 8 : then load context
MSRNE   spsr_cxsf, r14             ; 8 :    update spsr
LDMNEIA r13!, {r14, pc}^           ; 8 :    return
LDMIA   r13!, {r0-r3, r8}          ; 10 : else load reg
STMDB   r13!, {r0-r11}             ; 10 :   save context
BL      continue_servicing         ; 11 : continue service
LDMIA   r13!, {r0-r12, r14}        ; 12 : restore context
MSR     spsr_cxsf, r14             ; 12 : update spsr
LDMIA   r13!, {r14, pc}^           ; 12 : return
```

The start of the handler includes a normal interrupt entry point, with four being subtracted from the register *r14_irq*.

It is now important to assign values to the various fields in the data structure pointed to by register *r13_irq*. The registers that are recorded are *r14_irq*, *spsr_irq*, and *r0*. The register *r0* is used to transfer a pointer to the data structure when swapping to *SVC* mode since register *r0* will not be banked. This is why register *r13_irq* cannot be used for this purpose: it is not visible from *SVC* mode.

The pointer to the data structure is saved by copying register *r13_irq* into *r0*.

Offset (from *r13_irq*)	Value
+0	*r0* (on entry)
+4	*spsr_irq*
+8	*r14_irq*

The handler will now set the processor into *SVC* mode using the standard procedure of manipulating the *cpsr*. The link register *r14* for *SVC* mode is saved on the SVC stack. Subtracting 8 provides room on the stack for two 32-bit words.

Register *r14_irq* is then recovered and stored on the SVC stack. Now both the link registers *r14* for IRQ and SVC are stored on the SVC stack.

The rest of the IRQ context is recovered from the data structure passed into the *SVC* mode. Register *r14_svc* will now contain the *spsr* for *IRQ* mode.

Registers are then saved onto the SVC stack. Register *r8* is used to hold the interrupt mask for the interrupts that have been disabled in the interrupt handler. They will be reenabled later.

The interrupt source(s) are then disabled. An embedded system would at this point prioritize the interrupts and disable all interrupts lower than the current priority to prevent a low-priority interrupt from locking out a high-priority interrupt. Interrupt prioritizing will be discussed later on in this chapter.

Since the interrupt source has been cleared, it is now safe to reenable IRQ exceptions. This is achieved by clearing the *i* bit in the *cpsr*. Note that the interrupt controller still has external interrupts disabled.

It is now possible to process the interrupt. The interrupt processing should not attempt to do a context switch because the external source interrupt is disabled. If during the interrupt processing a context switch is needed, it should set a flag that could be picked up later by the interrupt handler. It is now safe to reenable external interrupts.

The handler needs to check if further processing is required. If the returned value is nonzero in register *r0*, then no further processing is required. If zero, the handler restores the context and then returns control back to the suspended task.

A stack frame now has to be created so that the service routine can complete. This is achieved by restoring parts of the context and then storing the complete context back on to the SVC stack.

The subroutine `continue_servicing`, which will complete the servicing of the interrupt, is called. This routine is not provided because it is specific to an implementation.

After the interrupt routine has been serviced, control can be given back to the suspended task.

SUMMARY **Reentrant Interrupt Handler**

- Handles multiple interrupts that can be prioritized.
- Low interrupt latency.
- Advantage: handles interrupts with differing priorities.
- Disadvantage: tends to be more complex.

9.3.4 Prioritized Simple Interrupt Handler

Both the nonnested interrupt handler and the nested interrupt handler service interrupts on a first-come-first-served basis. In comparison, the prioritized interrupt handler will associate a priority level with a particular interrupt source. The priority level is used to dictate the order that the interrupts will be serviced. Thus, a higher-priority interrupt will take precedence over a lower-priority interrupt, which is a particularly desirable characteristic in many embedded systems.

Methods of handling prioritization can either be achieved in hardware or software. For hardware prioritization, the handler is simpler to design since the interrupt controller will provide the current highest-priority interrupt that requires servicing. These systems require more initialization code at startup since the interrupts and associated priority level tables have to be constructed before the system can be switched on; software prioritization, on the other hand, requires the additional assistance of an external interrupt controller. This interrupt controller has to provide a minimal set of functions that include being able to set and un-setmasks, and to read the interrupt status and source.

The rest of this section will cover a software prioritization technique chosen because it is a general method and does not rely on a specialized interrupt controller. To help describe the priority interrupt handler, we will introduce a fictional interrupt controller based upon a standard interrupt controller from ARM. The controller takes multiple interrupt sources and generates an IRQ and/or FIQ signal depending upon whether a particular interrupt source is enabled or disabled.

Figure 9.11 shows a flow diagram of a simple priority interrupt handler, based on a reentrant interrupt handler.

EXAMPLE 9.11

The interrupt controller has a register (*IRQRawStatus*) that holds the *raw interrupt status*—the state of the interrupt signals prior to being masked by the controller. The *IRQEnable* register determines which interrupts are masked from the processor. This register can only be set or cleared using *IRQEnableSet* and *IRQEnableClear*. Table 9.10 shows the interrupt controller register names, offsets from the controller's base address, read/write operations, and a description of the registers.

```
I_Bit           EQU 0x80

PRIORITY_0      EQU 2                   ; Comms Rx
PRIORITY_1      EQU 1                   ; Comms Tx
PRIORITY_2      EQU 0                   ; Timer 1
PRIORITY_3      EQU 3                   ; Timer 2

BINARY_0        EQU 1<<PRIORITY_0       ; 1<<2 0x00000004
BINARY_1        EQU 1<<PRIORITY_1       ; 1<<1 0x00000002
BINARY_2        EQU 1<<PRIORITY_2       ; 1<<0 0x00000001
BINARY_3        EQU 1<<PRIORITY_3       ; 1<<3 0x00000008

MASK_3          EQU BINARY_3
MASK_2          EQU MASK_3+BINARY_2
MASK_1          EQU MASK_2+BINARY_1
MASK_0          EQU MASK_1+BINARY_0

ic_Base         EQU 0x80000000
IRQStatus       EQU 0x0
```

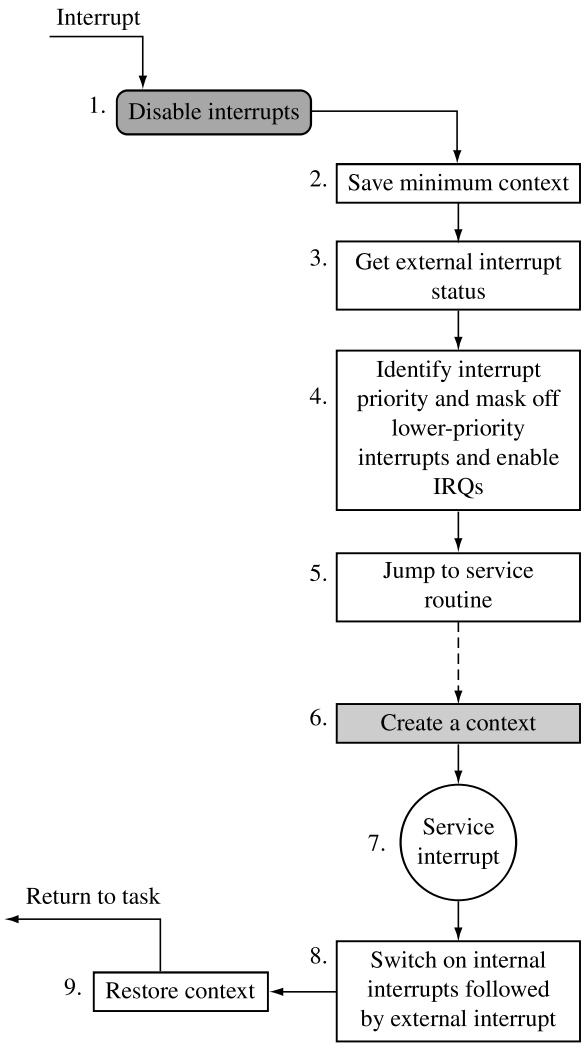


Figure 9.11 Priority interrupt handler.

```
IRQRawStatus    EQU 0x4
IRQEnable       EQU 0x8
IRQEnableSet    EQU 0x8
IRQEnableClear  EQU 0xc
```

Table 9.10 Interrupt controller registers.

Register	Offset	R/W	Description
IRQRawStatus	+0x04	r	represents status of the interrupt sources
IRQEnable	+0x08	r	masks the interrupt sources that generate IRQ/FIQ to the CPU
IRQStatus	+0x00	r	represents interrupt sources after masking
IRQEnableSet	+0x08	w	sets bits in the interrupt enable register
IRQEnableClear	+0x0c	w	clears bits in the interrupt enable register

```
IRQ_Handler        ; instruction            state : comment
        SUB     r14, r14, #4               ; 2 : r14_irq -= 4
        STMFD   r13!, {r14}                ; 2 : save r14_irq
        MRS     r14, spsr                  ; 2 : copy spsr_irq
        STMFD   r13!, {r10,r11,r12,r14}    ; 2 : save context
        LDR     r14, =ic_Base              ; 3 : int crtl addr
        MOV     r11, #PRIORITY_3           ; 3 : default priority
        LDR     r10, [r14, #IRQStatus]     ; 3 : load IRQ status
        TST     r10, #BINARY_3             ; 4 : if Timer 2
        MOVNE   r11, #PRIORITY_3           ; 4 :   then P3(lo)
        TST     r10, #BINARY_2             ; 4 : if Timer 1
        MOVNE   r11, #PRIORITY_2           ; 4 :   then P2
        TST     r10, #BINARY_1             ; 4 : if Comm Tx
        MOVNE   r11, #PRIORITY_1           ; 4 :   then P1
        TST     r10, #BINARY_0             ; 4 : if Comm Rx
        MOVNE   r11, #PRIORITY_0           ; 4 :  then P0(hi)
        LDR     r12, [r14,#IRQEnable]      ; 4 : IRQEnable reg
        ADR     r10, priority_masks        ; 4 : mask address
        LDR     r10, [r10,r11,LSL #2]      ; 4 : priority value
        AND     r12, r12,r10               ; 4 : AND enable reg
        STR     r12, [r14,#IRQEnableClear] ; 4 : disable ints
        MRS     r14, cpsr                  ; 4 : copy cpsr
        BIC     r14, r14, #I_Bit           ; 4 : clear I-bit
        MSR     cpsr_c, r14                ; 4 : enable IRQ ints
        LDR     pc, [pc, r11, LSL#2]       ; 5 : jump to an ISR
        NOP                                  ;
        DCD     service_timer1             ; timer1 ISR
        DCD     service_commtx             ; commtx ISR
        DCD     service_commrx             ; commrx ISR
        DCD     service_timer2             ; timer2 ISR

priority_masks
        DCD     MASK_2                     ; priority mask 2
```

```
        DCD     MASK_1                  ; priority mask 1
        DCD     MASK_0                  ; priority mask 0
        DCD     MASK_3                  ; priority mask 3
        ...
service_timer1
        STMFD   r13!, {r0-r9}           ; 6 : save context
        <service routine>
        LDMFD   r13!, {r0-r10}          ; 7 : restore context
        MRS     r11, cpsr               ; 8 : copy cpsr
        ORR     r11, r11, #I_Bit        ; 8 : set I-bit
        MSR     cpsr_c, r11             ; 8 : disable IRQ
        LDR     r11, =ic_Base           ; 8 : int ctrl addr
        STR     r12, [r11, #IRQEnableSet]  ; 8 : enable ints
        LDMFD   r13!, {r11, r12, r14}   ; 9 : restore context
        MSR     spsr_cxsf, r14          ; 9 : set spsr
        LDMFD   r13!, {pc}^             ; 9 : return
```

Most interrupt controllers also have a corresponding set of registers for the FIQ exceptions and even allow individual interrupt sources to be attached to a particular interrupt signal going to the core. Thus, by programming the controller, a particular interrupt source can be made to cause either an IRQ or FIQ exception.

The registers are offset from a base address in memory. Table 9.10 shows all the offsets for the various registers from interrupt controller base address `ic_Base`. Note that offset 0x08 is used for both `IRQEnable` and `IRQEnableSet`.

In the interrupt controller each bit is associated with a particular interrupt source (see Figure 9.12). For example, bit 2 is associated with a receive interrupt source for serial communication.

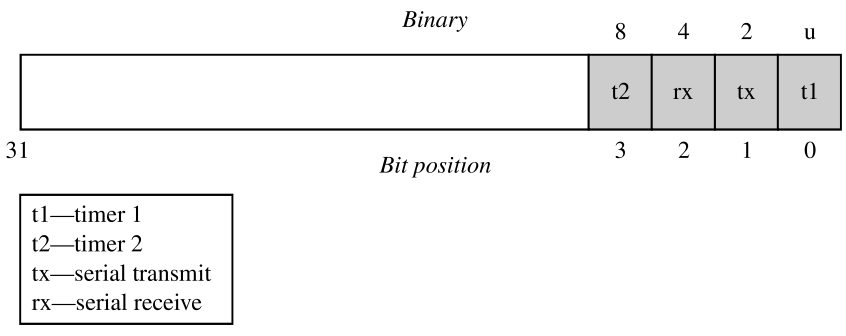


Figure 9.12 32-bit interrupt control register.

The PRIORITY_*x* defines the four interrupt sources, used in the example, to a corresponding set of priority levels, where PRIORITY_0 is the highest-priority interrupt and PRIORITY_3 is the lowest-priority interrupt.

The BINARY_*x* defines provide the bit patterns for each of the priority levels. For instance, for a PRIORITY_0 interrupt the binary pattern would be 0x00000004 (or $1 \ll 2$). For each priority level there is a corresponding mask that masks out all interrupts that are equal or lower in priority. For instance, MASK_2 will mask out interrupts from Timer2 (priority = 3) and CommRx (priority = 2).

The defines for the interrupt controller registers are also listed. ic_Base is the base address, and the remaining defines (for instance, IRQStatus) are all offsets from that base address.

The priority interrupt handler starts with a standard entry, but at first only the IRQ link register is stored onto the IRQ stack.

Next the handler obtains the *spsr* and places the contents into register *r14_irq* and frees up a group of registers for use in processing the prioritization.

The handler needs to obtain the status of the interrupt controller. This is achieved by loading in the base address of the interrupt controller into register *r14* and loading register *r10* with ic_Base (register *r14*) offset by IRQStatus (0x00).

The handler now needs to determine the highest-priority interrupt by testing the status information. If a particular interrupt source matches a priority level, then the priority level is set in register *r11*. The method compares the interrupt source with all the set priority levels, starting first with the lowest priority and working to the highest priority.

After this code fragment, register *r14_irq* will contain the base address of the interrupt controller, and register *r11* will contain the bit number of the highest-priority interrupt. It is now important to disable the lower- and equal-priority interrupts so that the higher-priority interrupts can still interrupt the handler.

Notice that this method is more deterministic since the time taken to discover the priority is always the same.

To set the interrupt mask in the controller, the handler must determine the current IRQ enable register and also obtain the start address of the priority mask table. The priority_masks are defined at the end of the handler.

Register *r12* will now contain the current IRQ enable register, and register *r10* will contain the start address of the priority table. To obtain the correct mask, register *r11* is shifted left by two (using the barrel shifter LSL #2). This will multiply the address by four and add that to the start address of the priority table.

Register *r10* contains the new mask. The next step is to clear the lower-priority interrupts using the mask, by performing a binary AND with the mask and register *r12* (*IRQEnable* register) and then clearing the bits by storing the new mask into *IRQEnableClear* register. It is now safe to enable IRQ exceptions by clearing the *i* bit in the *cpsr*.

Lastly the handler needs to jump to the correct service routine, by modifying register *r11* (which still contains the highest-priority interrupt) and the *pc*. Shifting register *r11* left by two (multiplying by four) and adding it to the *pc* allows the handler to jump to the correct routine by loading the address of the service routine directly into the *pc*.

The jump table has to follow the instruction that loads the *pc*. There is an `NOP` in between the jump table and the instruction that manipulates the *pc* because the *pc* will be pointing two instructions ahead (or eight bytes). The *priority mask table* is in interrupt source bit order.

Each ISR follows the same entry style. The example given is for the `timer1` interrupt service routine.

The ISR is then inserted after the header above. Once the ISR is complete, the interrupt sources must be reset and control passed back to the interrupted task.

The handler must disable the IRQs before the interrupts can be switched back on. The external interrupts can now be restored to their original value, which is possible because the service routine did not modify register *r12* and so it still contains the original value.

To return back to the interrupted task, context is restored and the original *spsr* is copied back into the *spsr_irq*.

SUMMARY **Prioritized Simple Interrupt Handler**

- Handles prioritized interrupts.
- Low interrupt latency.
- Advantage: deterministic interrupt latency since the priority level is identified first and then the service is called after the lower-priority interrupts are masked.
- Disadvantage: the time taken to get to a low-priority service routine is the same as for a high-priority routine.

9.3.5 Prioritized Standard Interrupt Handler

Following on from the prioritized simple interrupt handler, the next handler adds an additional level of complexity. The prioritized simple interrupt handler tested all the interrupts to establish the highest priority—an inefficient method of establishing the priority level but it does have the advantage of being deterministic since each interrupt priority will take the same length of time to be identified.

An alternative approach is to jump early when the highest-priority interrupt has been identified (see Figure 9.13), by setting the *pc* and jumping immediately once the priority level has been established. This means that the identification section of the code for the prioritized standard interrupt handler is more involved than for the prioritized simple interrupt handler. The identification section will determine the priority level and jump immediately to a routine that will handle the masking of the lower-priority interrupts and then jump again via a jump table to the appropriate ISR.

EXAMPLE 9.12 A prioritized standard interrupt handler starts the same as a prioritized simple interrupt handler but intercepts the interrupts with a higher-priority earlier. Register *r14* is assigned to point to the base of the interrupt controller and load register *r10* with the interrupt controller status register. To allow the handler to be relocatable, the current address pointed to by the *pc* is recorded into register *r11*.

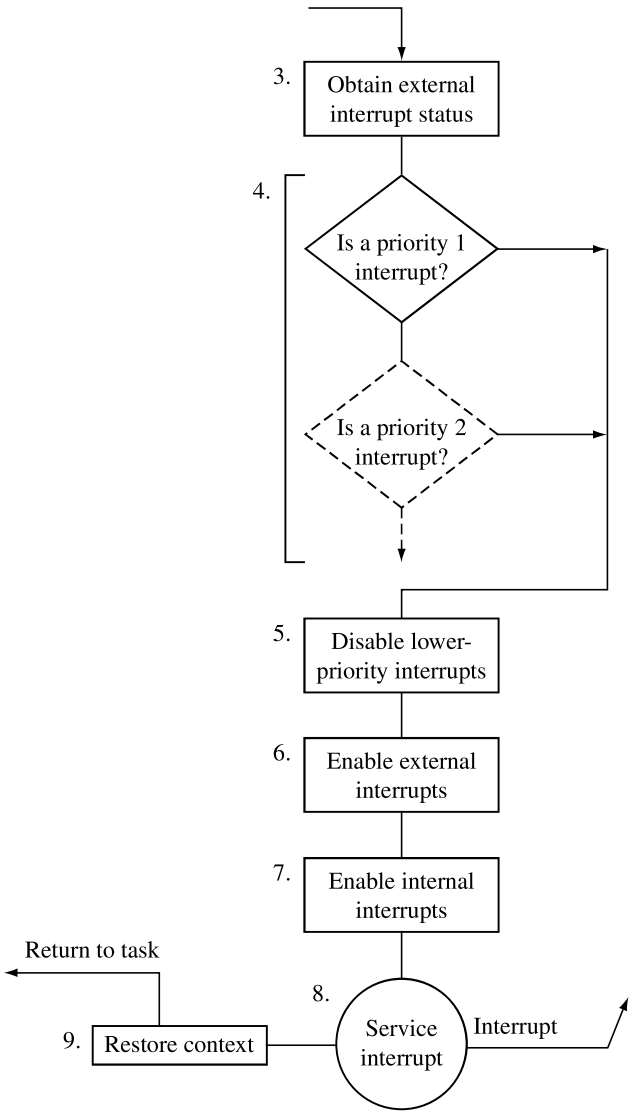


Figure 9.13 Part of a prioritized standard interrupt handler.

```
I_Bit              EQU 0x80

PRIORITY_0         EQU 2                    ; Comms Rx
PRIORITY_1         EQU 1                    ; Comms Tx
PRIORITY_2         EQU 0                    ; Timer 1
PRIORITY_3         EQU 3                    ; Timer 2

BINARY_0           EQU 1<<PRIORITY_0        ; 1<<2 0x00000004
BINARY_1           EQU 1<<PRIORITY_1        ; 1<<1 0x00000002
BINARY_2           EQU 1<<PRIORITY_2        ; 1<<0 0x00000001
BINARY_3           EQU 1<<PRIORITY_3        ; 1<<3 0x00000008

MASK_3             EQU BINARY_3
MASK_2             EQU MASK_3+BINARY_2
MASK_1             EQU MASK_2+BINARY_1
MASK_0             EQU MASK_1+BINARY_0

ic_Base            EQU 0x80000000
IRQStatus          EQU 0x0
IRQRawStatus       EQU 0x4
IRQEnable          EQU 0x8
IRQEnableSet       EQU 0x8
IRQEnableClear     EQU 0xc

IRQ_Handler     ; instruction            state : comment
        SUB     r14, r14, #4               ; 2 : r14_irq -= 4
        STMFD   r13!, {r14}                ; 2 : save r14_irq
        MRS     r14, spsr                  ; 2 : copy spsr_irq
        STMFD   r13!,{r10,r11,r12,r14}     ; 2 : save context
        LDR     r14, =ic_Base              ; 3 : int crtl addr
        LDR     r10, [r14, #IRQStatus]     ; 3 : load IRQ status
        MOV     r11, pc                    ; 4 : copy pc
        TST     r10, #BINARY_0             ; 5 : if CommRx
        BLNE    disable_lower              ; 5 :  then branch
        TST     r10, #BINARY_1             ; 5 : if CommTx
        BLNE    disable_lower              ; 5 :  then branch
        TST     r10, #BINARY_2             ; 5 : if Timer1
        BLNE    disable_lower              ; 5 :  then branch
        TST     r10, #BINARY_3             ; 5 : if Timer2
        BLNE    disable_lower              ; 5 :  then branch
disable_lower
        SUB     r11, r14, r11              ; 5 : r11=r14-copy of pc
        LDR     r12,=priority_table        ; 5 : priority table
```

```
        LDRB    r11,[r12,r11,LSR #3]        ; 5 : mem8[tbl+(r11>>3)]
        ADR     r10, priority_masks         ; 5 : priority mask
        LDR     r10, [r10,r11,LSL #2]       ; 5 : load mask
        LDR     r14, =ic_Base               ; 6 : int crtl addr
        LDR     r12, [r14,#IRQEnable]       ; 6 : IRQ enable reg
        AND     r12, r12, r10               ; 6 : AND enable reg
        STR     r12, [r14,#IRQEnableClear]  ; 6 : disable ints
        MRS     r14, cpsr                   ; 7 : copy cpsr
        BIC     r14, r14, #I_Bit            ; 7 : clear I-bit
        MSR     cpsr_c, r14                 ; 7 : enable IRQ
        LDR     pc, [pc, r11, LSL#2]        ; 8 : jump to an ISR
        NOP                                 ;

        DCD     service_timer1              ; timer1 ISR
        DCD     service_commtx              ; commtx ISR
        DCD     service_commrx              ; commrx ISR
        DCD     service_timer2              ; timer2 ISR

priority_masks
        DCD     MASK_2                      ; priority mask 2
        DCD     MASK_1                      ; priority mask 1
        DCD     MASK_0                      ; priority mask 0
        DCD     MASK_3                      ; priority mask 3

priority_table
        DCB     PRIORITY_0                  ; priority 0
        DCB     PRIORITY_1                  ; priority 1
        DCB     PRIORITY_2                  ; priority 2
        DCB     PRIORITY_3                  ; priority 3
        ALIGN
```

The interrupt source can now be tested by comparing the highest to the lowest priority. The first priority level that matches the interrupt source determines the priority level of the incoming interrupt because each interrupt has a preset priority level. Once a match is achieved, then the handler can branch to the routine that masks off the lower-priority interrupts.

To disable the equal- or lower-priority interrupts, the handler enters a routine that first calculates the priority level using the base address in register *r11* and link register *r14*.

Following the SUB instruction register *r11* will now contain the value 4, 12, 20, or 28. These values correspond to the priority level of the interrupt multiplied by eight plus four. Register *r11* is then divided by eight and added to the address of the *priority_table*. Following the LDRB register *r11* will equal one of the priority interrupt numbers (0, 1, 2, or 3).

The priority mask can now be determined, using the technique of shifting left by two and adding that to the register *r10*, which contains the address of the *priority_mask*.

The base address for the interrupt controller is copied into register *r14_irq* and is used to obtain the *IRQEnable* register in the controller and place it into register *r12*.

Register *r10* contains the new mask. The next step is to clear the lower-priority interrupts using this mask by performing a binary `AND` with the mask and *r12* (*IRQEnable* register) and storing the result into the *IRQEnableClear* register. It is now safe to enable IRQ exceptions by clearing the *i* bit in the *cpsr*.

Lastly the handler needs to jump to the correct service routine, by modifying *r11* (which still contains the highest-priority interrupt) and the *pc*. Shifting register *r11* left by two (multiplying *r11* by four) and adding it to the *pc* allows the handler to jump to the correct routine by loading the address of the service routine directly into the *pc*. The jump table must follow the instruction that loads the *pc*. There is an `NOP` between the jump table and the `LDR` instruction that modifies the *pc* because the *pc* is pointing two instructions ahead (or eight bytes).

Note that the priority mask table is in interrupt bit order, and the priority table is in priority order.

SUMMARY **Prioritized Standard Interrupt Handler**

- Handles higher-priority interrupts in a shorter time than lower-priority interrupts.
- Low interrupt latency.
- Advantage: higher-priority interrupts treated with greater urgency with no duplication of code to set external interrupt masks.
- Disadvantage: there is a time penalty since this handler requires two jumps, resulting in the pipeline being flushed each time a jump occurs.

9.3.6 Prioritized Direct Interrupt Handler

One difference between the prioritized direct interrupt handler and the prioritized standard interrupt handler is that some of the processing is moved out of the handler into the individual ISRs. The moved code masks out the lower-priority interrupts. Each ISR will have to mask out the lower-priority interrupts for the particular priority level, which can be a fixed number since the priority level has already been previously determined.

The second difference is that the prioritized direct interrupt handler jumps directly to the appropriate ISR. Each ISR is responsible for disabling the lower-priority interrupts before modifying the *cpsr* to reenable interrupts. This type of handler is relatively simple since the masking is done by the individual ISR, but there is a small amount of code duplication since each interrupt service routine is effectively carrying out the same task.

EXAMPLE 9.13 The `bit_x` defines associate an interrupt source with a bit position within the interrupt controller, which will be used to help mask the lower-priority interrupts within an ISR.

Once the context is saved, the base address of the ISR table has to be loaded into register *r12*. This register is used to jump to the correct ISR once the priority has been established for the interrupt source.

```
I_Bit           EQU 0x80

PRIORITY_0      EQU 2                       ; Comms Rx
PRIORITY_1      EQU 1                       ; Comms Tx
PRIORITY_2      EQU 0                       ; Timer 1
PRIORITY_3      EQU 3                       ; Timer 2

BINARY_0        EQU 1<<PRIORITY_0           ; 1<<2 0x00000004
BINARY_1        EQU 1<<PRIORITY_1           ; 1<<1 0x00000002
BINARY_2        EQU 1<<PRIORITY_2           ; 1<<0 0x00000001
BINARY_3        EQU 1<<PRIORITY_3           ; 1<<3 0x00000008

MASK_3          EQU BINARY_3
MASK_2          EQU MASK_3+BINARY_2
MASK_1          EQU MASK_2+BINARY_1
MASK_0          EQU MASK_1+BINARY_0

ic_Base         EQU 0x80000000
IRQStatus       EQU 0x0
IRQRawStatus    EQU 0x4
IRQEnable       EQU 0x8
IRQEnableSet    EQU 0x8
IRQEnableClear  EQU 0xc

bit_timer1      EQU 0
bit_commtx      EQU 1
bit_commrx      EQU 2
bit_timer2      EQU 3

IRQ_Handler     ; instruction               comment
        SUB     r14, r14, #4                ; r14_irq-=4
        STMFD   r13!, {r14}                 ; save r14_irq
        MRS     r14, spsr                   ; copy spsr_irq
        STMFD   r13!,{r10,r11,r12,r14}      ; save context
        LDR     r14, =ic_Base               ; int crtl addr
        LDR     r10, [r14, #IRQStatus]      ; load IRQ status
        ADR     r12, isr_table              ; obtain ISR table
        TST     r10, #BINARY_0              ; if CommRx
        LDRNE   pc, [r12, #PRIORITY_0<<2]   ;   then CommRx ISR
```

```
        TST     r10, #BINARY_1              ; if CommTx
        LDRNE   pc, [r12, #PRIORITY_1<<2]   ;   then CommTx ISR
        TST     r10, #BINARY_2              ; if Timer1
        LDRNE   pc, [r12, #PRIORITY_2<<2]   ;   then Timer1 ISR
        TST     r10, #BINARY_3              ; if Timer2
        LDRNE   pc, [r12, #PRIORITY_3<<2]   ;   then Timer2 ISR
        B       service_none

isr_table
        DCD     service_timer1              ; timer1 ISR
        DCD     service_commtx              ; commtx ISR
        DCD     service_commrx              ; commrx ISR
        DCD     service_timer2              ; timer2 ISR

priority_masks
        DCD     MASK_2                      ; priority mask 2
        DCD     MASK_1                      ; priority mask 1
        DCD     MASK_0                      ; priority mask 0
        DCD     MASK_3                      ; priority mask 3
        ...
service_timer1
        MOV     r11, #bit_timer1            ; copy bit_timer1
        LDR     r14, =ic_Base               ; int ctrl addr
        LDR     r12, [r14,#IRQEnable]       ; IRQ enable register
        ADR     r10, priority_masks         ; obtain priority addr
        LDR     r10, [r10,r11,LSL#2]        ; load priority mask
        AND     r12, r12, r10               ; AND enable reg
        STR     r12, [r14, #IRQEnableClear] ; disable ints
        MRS     r14, cpsr                   ; copy cpsr
        BIC     r14, r14, #I_Bit            ; clear I-bit
        MSR     cpsr_c, r14                 ; enable IRQ
        <rest of the ISR>
```

The priority interrupt is established by checking the highest-priority interrupt first and then working down to the lowest. Once a priority interrupt is identified, the *pc* is then loaded with the address of the appropriate ISR. The indirect address is stored at the address of the *isr_table* plus the priority level shifted two bits to the left (multiplied by four). Alternatively you could use a conditional branch BNE.

The ISR jump table *isr_table* is ordered with the highest-priority interrupt at the beginning of the table.

The *service_timer1* entry shows an example of an ISR used in a priority direct interrupt handler. Each ISR is unique and depends upon the particular interrupt source.

A copy of the base address for the interrupt controller is placed into register *r14_irq*. This address plus an offset is used to copy the *IRQEnable* register into register *r12*.

The address of the priority mask table has to be copied into register *r10* so it can be used to calculate the address of the actual mask. Register *r11* is shifted left two positions, which gives an offset of 0, 4, 8, or 12. The offset plus the address of the priority mask table address is used to load the mask into register *r10*. The priority mask table is the same as for the priority interrupt handler in the previous section.

Register *r10* will contain the ISR mask, and register *r12* will contain the current mask. A binary `AND` is used to merge the two masks. Then the new mask is used to configure the interrupt controller using the *IRQEnableClear* register. It is now safe to enable IRQ exceptions by clearing the *i* bit in the *cpsr*.

The handler can continue servicing the current interrupt unless an interrupt with a higher priority occurs, in which case that interrupt will take precedence over the current interrupt.

SUMMARY **Prioritized Direct Interrupt Handler**

- Handles higher-priority interrupts in a shorter time. Goes directly to the specific ISR.
- Low interrupt latency.
- Advantage: uses a single jump and saves valuable cycles to go to the ISR.
- Disadvantage: each ISR has a mechanism to set the external interrupt mask to stop lower-priority interrupts from halting the current ISR, which adds extra code to each ISR.

9.3.7 PRIORITIZED GROUPED INTERRUPT HANDLER

Lastly, the prioritized grouped interrupt handler differs from the other prioritized interrupt handlers since it is designed to handle a large set of interrupts. This is achieved by grouping interrupts together and forming a subset, which can then be given a priority level.

The designer of an embedded system must identify each subset of interrupt sources and assign a group priority level to that subset. It is important to be careful when selecting the subsets of interrupt sources since the groups can determine the characteristics of the system. Grouping the interrupt sources together tends to reduce the complexity of the handler since it is not necessary to scan through every interrupt to determine the priority level. If a prioritized grouped interrupt handler is well designed, it will dramatically improve overall system response times.

EXAMPLE 9.14 This handler has been designed to have two priority groups. Timer sources are grouped into group 0, and communication sources are grouped into group 1 (see Table 9.11.) Group 0 interrupts are given a higher priority than group 1 interrupts.

```
I_Bit           EQU 0x80

PRIORITY_0      EQU 2                   ; Comms Rx
```

Table 9.11 Group interrupt sources.

Group	Interrupts
0	timer1, timer2
1	commtx, commrx

```
PRIORITY_1      EQU 1                ; Comms Tx
PRIORITY_2      EQU 0                ; Timer 1
PRIORITY_3      EQU 3                ; Timer 2

BINARY_0        EQU 1<<PRIORITY_0     ; 1<<2 0x00000004
BINARY_1        EQU 1<<PRIORITY_1     ; 1<<1 0x00000002
BINARY_2        EQU 1<<PRIORITY_2     ; 1<<0 0x00000001
BINARY_3        EQU 1<<PRIORITY_3     ; 1<<3 0x00000008

GROUP_0         EQU BINARY_2|BINARY_3
GROUP_1         EQU BINARY_0|BINARY_1

GMASK_1         EQU GROUP_1
GMASK_0         EQU GMASK_1+GROUP_0

MASK_TIMER1     EQU GMASK_0
MASK_COMMTX     EQU GMASK_1
MASK_COMMRX     EQU GMASK_1
MASK_TIMER2     EQU GMASK_0

ic_Base         EQU 0x80000000
IRQStatus       EQU 0x0
IRQRawStatus    EQU 0x4
IRQEnable       EQU 0x8
IRQEnableSet    EQU 0x8
IRQEnableClear  EQU 0xc

interrupt_handler
        SUB     r14, r14,#4              ; r14_irq-=4
        STMFD   r13!, {r14}              ; save r14_irq
        MRS     r14, spsr                ; copy spsr_irq
        STMFD   r13!, {r10,r11,r12,r14}  ; save context
        LDR     r14, =ic_Base            ; int ctrl addr
        LDR     r10, [r14, #IRQStatus]   ; load IRQ status
        ANDS    r11, r10, #GROUP_0       ; belong to GROUP_0
        ANDEQS  r11, r10, #GROUP_1       ; belong to GROUP_1
```

```
        AND     r10, r11, #0xf              ; mask off top 24-bit
        ADR     r11, lowest_significant_bit ; load LSB addr
        LDRB    r11, [r11, r10]             ; load byte
        B       disable_lower_priority      ; jump to routine

lowest_significant_bit
        ;       0    1 2 3 4 5 6 7 8 9 a b c d e f
        DCB     0xff,0,1,0,2,0,1,0,3,0,1,0,2,0,1,0

disable_lower_priority
        CMP     r11, #0xff                  ; if unknown
        BEQ     unknown_condition           ;   then jump
        LDR     r12, [r14, #IRQEnable]      ; load IRQ enable reg
        ADR     r10, priority_mask          ; load priority addr
        LDR     r10, [r10, r11, LSL #2]     ; mem32[r10+r11<<2]
        AND     r12, r12, r10               ; AND enable reg
        STR     r12, [r14, #IRQEnableClear] ; disable ints
        MRS     r14, cpsr                   ; copy cpsr
        BIC     r14, r14, #I_Bit            ; clear I-bit
        MSR     cpsr_c, r14                 ; enable IRQ ints
        LDR     pc, [pc, r11, LSL #2]       ; jump to an ISR
        NOP
        DCD     service_timer1              ; timer1 ISR
        DCD     service_commtx              ; commtx ISR
        DCD     service_commrx              ; commrx ISR
        DCD     service_timer2              ; timer2 ISR

priority_mask
        DCD     MASK_TIMER1                 ; mask GROUP 0
        DCD     MASK_COMMTX                 ; mask GROUP 1
        DCD     MASK_COMMRX                 ; mask GROUP 1
        DCD     MASK_TIMER2                 ; mask GROUP 0
```

The GROUP_*x* defines assign the various interrupt sources to their specific priority level by using a binary OR operation on the binary patterns. The GMASK_*x* defines assign the masks for the grouped interrupts. The MASK_*x* defines connect each GMASK_*x* to a specific interrupt source, which can then be used in the priority mask table.

After the context has been saved the interrupt handler loads the IRQ status register using an offset from the interrupt controller base address.

The handler then identifies the group to which the interrupt source belongs by using the binary AND operation on the source. The letter *S* postfixed to the instructions means update condition flags in the *cpsr*.

Register *r11* will now contain the highest-priority group 0 or 1. The handler now masks out the other interrupt sources by applying a binary AND operation with 0xf.

Table 9.12 Lowest significant bit table.

Binary pattern	Value
0000	unknown
0001	0
0010	1
0011	0
0100	2
0101	0
0110	1
0111	0
1000	3
1001	0
1010	1
1011	0
1100	2
1101	0
1110	1
1111	0

The address of the lowest significant bit table is then loaded into register *r11*. A byte is loaded from the start of the table using the value in register *r10* (0, 1, 2, or 3, see Table 9.12). Once the lowest significant bit position is loaded into register *r11*, the handler branches to a routine.

The *disable_lower_priority* interrupt routine first checks for a spurious (no longer present) interrupt. If the interrupt is spurious, then the *unknown_condition* routine is called. The handler then loads the *IRQEnable* register and places the result in register *r12*.

The priority mask is found by loading in the address of the priority mask table and then shifting the data in register *r11* left by two. The result, 0, 4, 8, or 12, is added to the priority mask address. Register *r10* then contains a mask to disable the lower-priority group interrupts from being raised.

The next step is to clear the lower-priority interrupts using the mask by performing a binary `AND` with the mask in registers *r10* and *r12* (`IRQEnable` register) and then clearing the bits by saving the result into the `IRQEnableClear` register. At this point it is now safe to enable IRQ exceptions by clearing the *i* bit in the *cpsr*.

Lastly the handler jumps to the correct interrupt service routine by modifying register *r11* (which still contains the highest-priority interrupt) and the *pc*. By shifting register *r11* left by two and adding the result to the *pc* the address of the ISR is determined. This address is then loaded directly into the *pc*. Note that the jump table must follow the `LDR` instruction. The `NOP` is present due to the ARM pipeline.

SUMMARY **Prioritized Grouped Interrupt Handler**

- Mechanism for handling interrupts that are grouped into different priority levels.
- Low interrupt latency.
- Advantage: useful when the embedded system has to handle a large number of interrupts, and also reduces the response time since the determining of the priority level is shorter.
- Disadvantage: determining how the interrupts are grouped together.

9.3.8 VIC PL190 Based Interrupt Service Routine

To take advantage of the vector interrupt controller, the IRQ vector entry has to be modified.

```
0x00000018   LDR    pc,[pc,#-0xff0]  ; IRQ pc=mem32[0xfffff030]
```

This instruction loads an ISR address from the memory mapped location 0xffffff030 into the *pc* which bypasses any software interrupt handler since the interrupt source can be obtained directly from the hardware. It also reduces interrupt latency since there is only a single jump to a specific ISR.

Here is an example of VIC service routine:

```
INTON           EQU 0x0000              ; enable interrupts
SYS32md         EQU 0x1f                ; system mode
IRQ32md         EQU 0x12                ; IRQ mode
I_Bit           EQU 0x80
VICBaseAddr     EQU 0xfffff000          ; addr of VIC ctrl
VICVectorAddr   EQU VICBaseAddr+0x30    ; isr address of int

vector_service_routine
        SUB     r14,r14,#4              ; r14-=4
        STMFD   r13!, {r0-r3,r12,r14}   ; save context
        MRS     r12, spsr               ; copy spsr
        STMFD   r13!,{r12}              ; save spsr
        <clear the interrupt source>
        MSR     cpsr_c, #INTON|SYS32md  ; cpsr_c=ift_sys
        <interrupt service code>
        MSR     cpsr_c, #I_Bit|IRQ32md  ; cpsr_c=Ift_irq
        LDMFD   r13!, {r12}             ; restore (spsr_irq)
        MSR     spsr_cxsf, r12          ; restore spsr
        LDR     r1,=VICVectorAddr       ; load VectorAddress
        STR     r0, [r1]                ; servicing complete
        LDMFD   r13!, {r0-r3,r12,pc}^   ; return
```

This routine saves the context and *s psr_irq* before clearing the interrupt source. Once this is complete, the IRQ exceptions can be reenabled by clearing the *i* bit, and the processor mode is set to *system* mode. The service routine can then process the interrupt in *system* mode. Once complete, the IRQ exceptions are disabled by setting the *i* bit, and the processor mode is switched back to *IRQ* mode.

The *spsr_irq* is restored from the IRQ stack, preparing the routine to return to the interrupted task.

The service routine then writes to the `VICVectorAddr` register in the controller. Writing to this address indicates to the priority hardware that the interrupt has been serviced.

Note that since the VIC is basically a hardware interrupt handler, the array of ISR addresses must be preprogrammed into the VIC before it is activated.

9.4 Summary

An exception changes the normal sequential execution of instructions. There are seven exceptions: Data Abort, Fast Interrupt Request, Interrupt Request, Prefetch Abort, Software Interrupt, Reset, and Undefined Instruction. Each exception has an associated ARM processor mode. When an exception is raised, the processor goes into a specific mode and branches to an entry in the vector table. Each exception also has a priority level.

Interrupts are a special type of exception that are caused by an external peripheral. The IRQ exception is used for general operating system activities. The FIQ exception is normally reserved for a single interrupt source. *Interrupt latency* is the interval of time from an external interrupt request signal being raised to the first fetch of an instruction of a specific interrupt service routine (ISR).

We covered eight interrupt handling schemes, from a very simple nonnested interrupt handler that handles and services individual interrupts, to an advanced prioritized grouped interrupt handler that handles interrupts that are grouped into different priority levels.

This Page Intentionally Left Blank

CHAPTER 10
FIRMWARE

This chapter discusses firmware for ARM-based embedded systems. Firmware is an important part of any embedded system since it is frequently the first code to be ported and executed on a new platform. Firmware can vary from being a complete software embedded system to just a simple initialization and bootloader routine. We have divided this chapter into two sections.

The first section introduces firmware. In this section we define the term *firmware* and describe two popular industry standard firmware packages available for the ARM processor—ARM Firmware Suite and Red Hat's RedBoot. These firmware packages are general purpose and can be ported to different ARM platforms relatively easily and quickly.

The second section focuses on just the initialization and bootloader process. To help with this, we have developed a simple example called Sandstone. Sandstone is designed to initialize hardware, load an image into memory, and relinquish control of the *pc* over to that image.

We start by first discussing firmware and introduce the two common ARM firmware packages.

10.1 FIRMWARE AND BOOTLOADER

We realize that the use of terms may differ among engineers, but we will use the following definitions:

- The *firmware* is the deeply embedded, low-level software that provides an interface between the hardware and the application/operating system level software. It resides in the ROM and executes when power is applied to the embedded hardware system. Firmware can remain active after system initialization and supports basic

system operations. The choice of which firmware to use for a particular ARM-based system depends upon the specific application, which can range from loading and executing a sophisticated operating system to simply relinquishing control to a small microkernel. Consequently, requirements can vary greatly from one firmware implementation to another. For example, a small system may require just minimal firmware support to boot a small operating system. One of the main purposes of firmware is to provide a stable mechanism to load and boot an operating system.

- The *bootloader* is a small application that installs the operating system or application onto a hardware target. The bootloader only exists up to the point that the operating system or application is executing, and it is commonly incorporated into the firmware.

To help understand the features of different firmware implementations, we have a common execution flow (see Table 10.1). Each stage is now discussed in more detail.

The first stage is to set up the target platform—in other words, prepare the environment to boot an operating system since an operating system expects a particular type of environment before it can operate. This step involves making sure that the platform is correctly initialized (for example, making sure that the control registers of a particular microcontroller are placed at a known address or changing the memory map to an expected layout).

It is common for the same executable to operate on different cores and platforms. In this case, the firmware has to identify and discover the exact core and platform it is operating on. The core is normally recognized by reading register 0 in coprocessor 15, which holds both the processor type and the manufacturer name. There are multiple ways to identify the platform, from checking for the existence of a set of particular peripherals to simply reading a preprogrammed chip.

Table 10.1 Firmware execution flow.

Stage	Features
Set up target platform	Program the hardware system registers Platform identification Diagnostics Debug interface Command line interpreter
Abstract the hardware	Hardware Abstraction Layer Device driver
Load a bootable image	Basic filing system
Relinquish control	Alter the *pc* to point into the new image

Diagnostics software provides a useful way for quickly identifying basic hardware malfunctions. Because of the nature of this type of software, it tends to be specific to a particular piece of hardware.

Debug capabiliy is provided in the form of a module or monitor that provides software assistance for debugging code running on a hardware target. This assistance includes the following:

- Setting up breakpoints in RAM. A breakpoint allows a program to be interrupted and the state of the processor core to be examined.
- Listing and modifying memory (using peek and poke operations).
- Showing current processor register contents.
- Disassembling memory into ARM and Thumb instruction mnemonics.

These are interactive functions: you can either send the commands through a command line interpreter (CLI) or through a dedicated host debugger attached to the target platform. Unless the firmware has access to the internal hardware debug circuitry, only RAM images can be debugged through a software debug mechanism.

The CLI is commonly available on the more advanced firmware implementations. It allows you to change the operating system to be booted by altering the default configurations through typing commands at a command prompt. For embedded systems, the CLI is commonly controlled through a host terminal application. Communication between the host and the target is normally over a serial or network connection.

The second stage is to abstract the hardware. The *Hardware Abstraction Layer* (HAL) is a software layer that hides the underlying hardware by providing a set of defined programming interfaces. When you move to a new target platform, these programming interfaces remain constant but the underlying implementation changes. For instance, two target platforms might use a different timer peripheral. Each peripheral would require new code to initialize and configure the device. The HAL programming interface would remain unaltered even though both the hardware and software may differ greatly between implementations.

The HAL software that communicates with specific hardware peripherals is called a *device driver*. A device driver provides a standard application programming interface (API) to read and write to a specific peripheral.

The third stage is to load a bootable image. The ability of firmware to carry out this activity depends upon the type of media used to store the image. Note that not all operating system images or application images need to be copied into RAM. The operating system image or application image can simply execute directly from ROM.

ARM processors are normally found in small devices that include flash ROM. A common feature is a simple flash ROM filing system (FFS), which allows multiple executable images to be stored.

Other media devices, such as hard drives, require that the firmware incorporates a device driver that is suitable for accessing the hardware. Accessing the hardware requires

that the firmware has knowledge of the underlying filing system format, which gives the firmware the ability to read the filing system, find the file that contains the image, and copy the image into memory. Similarly, if the image is on the network, then the firmware must also understand the network protocol as well as the Ethernet hardware.

The load process has to take into account the image format. The most basic image format is plain binary. A plain binary image does not contain any header or debug information. A popular image format for ARM-based systems is Executable and Linking Format (ELF). This format was originally developed for UNIX systems and replaced the older format called Common Object File Format (COFF). ELF files come in three forms: relocatable, executable, and shared object.

Most firmware systems must deal with the executable form. Loading an ELF image involves deciphering the standard ELF header information (that is, execution address, type, size, and so on). The image may also be encrypted or compressed, in which case the load process would involve performing decryption or decompression on the image.

The fourth stage is to relinquish control. This is where the firmware hands over control of the platform to an operating system or application. Note that not all firmware hands over control; instead the firmware can remain the controlling software on the platform.

Firmware designed to pass control to an operating system may become inactive once the operating system has control. Alternatively, the Machine Independent Layer (MIL) or Hardware Abstraction Layer (HAL) part of the firmware can remain active. This layer exposes, through the SWI mechanism, a standard application interface for specific hardware devices.

Relinquishing control on an ARM system means updating the vector table and modifying the *pc*. Updating the vector table involves modifying particular exception and interrupt vectors so that they point to specialized operating system handlers. The *pc* has to be modified so that it points to the operating system entry point address.

For more sophisticated operating systems, such as Linux, relinquishing control requires that a standard data structure be passed to the kernel. This data structure explains the environment that the kernel will be running in. For example, one field may include the amount of available RAM on the platform, while another field includes the type of MMU being used.

We use these definitions to describe two common firmware suites.

10.1.1 ARM Firmware Suite

ARM has developed a firmware package called the ARM Firmware Suite (AFS). AFS is designed purely for ARM-based embedded systems. It provides support for a number of boards and processors including the Intel XScale and StrongARM processors. The package includes two major pieces of technology, a Hardware Abstraction Layer called μHAL (pronounced micro-HAL) and a debug monitor called Angel.

μHAL provides a low-level device driver framework that allows it to operate over different communication devices (for example, USB, Ethernet, or serial). It also provides a

standard API. Consequently, when a port takes place, the various hardware-specific parts must be implemented in accordance with the various μHAL API functions.

This has the advantage of making the porting process relatively straightforward since you have a standard function framework to work within. Once the firmware is ported, the task of moving an operating system over to the new target platform can take place. The speed of this activity depends upon whether the OS takes advantage of the ported μHAL API call to access the hardware.

μHAL supports these main features:

- *System initialization*—setting up the target platform and processor core. Depending upon the complexity of the target platform, this can either be a simple or complicated task.
- *Polled serial driver*—used to provide a basic method of communication with a host.
- *LED support*—allows control over the LEDs for simple user feedback. This provides an application the ability to display operational status.
- *Timer support*—allows a periodic interrupt to be set up. This is essential for preemptive context switching operating systems that require this mechanism.
- *Interrupt controllers*—support for different interrupt controllers.

The boot monitor in μHAL contains a CLI.

The second technology, Angel, allows communication between a host debugger and a target platform. It allows you to inspect and modify memory, download and execute images, set breakpoints, and display processor register contents. All this control is through the host debugger. The Angel debug monitor must have access to the SWI and IRQ or FIQ vectors.

Angel uses SWI instructions to provides a set of APIs that allow a program to open, read, and write to a host filing system. IRQ/FIQ interrupts are used for communication purposes with the host debugger.

10.1.2 Red Hat RedBoot

RedBoot is a firmware tool developed by Red Hat. It is provided under an open source license with no royalties or up front fees. RedBoot is designed to execute on different CPUs (for instance, ARM, MIPS, SH, and so on). It provides both debug capability through GNU Debugger (GDB), as well as a bootloader. The RedBoot software core is based on a HAL.

RedBoot supports these main features:

- *Communication*—configuration is over serial or Ethernet. For serial, X-Modem protocol is used to communicate with the GNU Debugger (GDB). For Ethernet, TCP is used to communicate with GDB. RedBoot supports a range of network standards, such as *bootp*, *telnet*, and *tftp*.

- *Flash ROM memory management*—provides a set of filing system routines that can download, update, and erase images in flash ROM. In addition, the images can either be compressed or uncompressed.
- *Full operating system support*—supports the loading and booting of Embedded Linux, Red Hat eCos, and many other popular operating systems. For Embedded Linux, RedBoot supports the ability to define parameters that are passed directly to the kernel upon booting.

10.2 Example: Sandstone

We have designed Sandstone to be a minimal system. It carries out only the following tasks: set up target platform environment, load a bootable image into memory, and relinquish control to an operating system. It is, however, still a real working example.

The implementation is specific to the ARM Evaluator-7T platform, which includes an ARM7TDMI processor. This example shows you exactly how a simple platform can be set up and a software payload can be loaded into memory and booted. The payload can either be an application or operating system image. Sandstone is a static design and cannot be configured after the build process is complete. Table 10.2 lists the basic characteristics of Sandstone.

We will walk you through the directory layout and code structure. The directory layout shows you where the source code is located and where the different build files are placed. The code structure focuses more on the actual initialization and boot process.

Note that Sandstone is written entirely in ARM assembler and is a working piece of code that can be used to intialize target hardware and boot any piece of software, within reason, on the ARM Evaluator-7T.

10.2.1 Sandstone Directory Layout

Sandstone can be found on our Web site. If you take a look at Sandstone, you will see that the directory structure is as shown in Figure 10.1. The structure follows a standard style

Table 10.2 Summary of Sandstone.

Feature	Configuration
Code	ARM instructions only
Tool chain	ARM Developer Suite 1.2
Image size	700 bytes
Source	17 KB
Memory	remapped

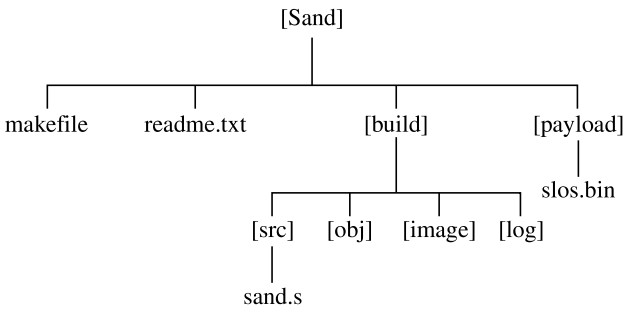


Figure 10.1 Standstone directory layout.

that we will continue to use in further chapters. The sandstone source file `sand.s` is located under the `sand/build/src` directory.

The object file produced by the assembler is placed under the `build/obj` directory. The object file is then linked, and the final Sandstone image is placed under the `sand/build/image` directory. This image includes both the Sandstone code and the payload. The payload image, the image that is loaded and booted by Sandstone, is found under the `sand/payload` directory.

For information about the Sandstone build procedure, take a look at the `readme.txt` file under the `sand` directory. This file contains a description of how to build the example binary image for the ARM Evaluator-7T.

10.2.2 Sandstone Code Structure

Sandstone consists of a single assembly file. The file structure is broken down into a number of steps, where each step corresponds to a stage in the execution flow of Sandstone (see Table 10.3).

Table 10.3 Sandstone execution flow.

Step	Description
1	Take the Reset exception
2	Start initializing the hardware
3	Remap memory
4	Initialize communication hardware
5	Bootloader—copy payload and relinquish control

We will take you through these steps, trying to avoid as much as possible the platform-specific parts. You should note that some specific parts are unavoidable (for example, configuring system registers and memory remapping).

The initial goal of Sandstone is to set up the target platform environment so that it can provide some form of feedback to indicate that the firmware is running and has control of the platform.

10.2.2.1 Step 1: Take the Reset Exception

Execution begins with a Reset exception. Only the reset vector entry is required in the default vector table. It is the very first instruction executed. You can see from the code here that all the vectors, apart from the reset vector, branch to a unique *dummy handler*—a branch instruction that causes an infinite loop. It is assumed that no exception or interrupt will occur during the operation of Sandstone. The reset vector is used to move the execution flow to the second stage.

```
        AREA start,CODE,READONLY
        ENTRY

sandstone_start
        B       sandstone_init1       ; reset vector
        B       ex_und                ; undefined vector
        B       ex_swi                ; swi vector
        B       ex_pabt               ; prefetch abort vector
        B       ex_dabt               ; data abort vector
        NOP                           ; not used...
        B       int_irq               ; irq vector
        B       int_fiq               ; fiq vector

ex_und  B       ex_und                ; loop forever
ex_swi  B       ex_swi                ; loop forever
ex_dabt B       ex_dabt               ; loop forever
ex_pabt B       ex_pabt               ; loop forever
int_irq B       int_irq               ; loop forever
int_fiq B       int_fiq               ; loop forever
```

`sandstone_start` is located at address `0x00000000`.

The results of executing step 1 are the following:

- Dummy handlers are set up.
- Control is passed to code to initialize the hardware.

10.2.2.2 Step 2: Start Initializing the Hardware

The primary phase in initializing hardware is setting up system registers. These registers have to be set up before accessing the hardware. For example, the ARM Evaluator-7T has a seven-segment display, which we have chosen to be used as a feedback tool to indicate that the firmware is active. Before we can set up the segment display, we have to position the base address of the system registers to a known location. In this case, we have picked the default address `0x03ff0000`, since this places all the hardware system registers away from both ROM and RAM, separating the peripherals and memory.

Consequently, all the microcontroller memory-mapped registers are located as an offset from `0x03ff0000`. This is achieved using the following code:

```
sandstone_init1
        LDR     r3, =SYSCFG       ; where SYSCFG=0x03ff0000
        LDR     r4, =0x03ffffa0
        STR     r4, [r3]
```

Register *r3* contains the default system register base address and is used to set the new default address, as well as other specific attributes such as the cache. Register *r4* contains the new configuration. The top 16 bits contain the high address of the new system register base address `0x03ff`, and the lower 16 bits contain the new attribute settings `0xffa0`.

After setting up the system register base address, the segment display can be configured. The segment display hardware is used to show Sandstone's progress. Note that the segment display is not shown since it is hardware specific.

The results of executing step 2 are the following:

- The system registers are set from a known base address—`0x03ff0000`.
- The segment display is configured, so that it can be used to display progress.

10.2.2.3 Step 3: Remap Memory

One of the major activities of hardware initialization is to set up the memory environment. Sandstone is designed to initialize SRAM and remap memory. This process occurs fairly early on in the initialization of the system. The platform starts in a known memory state, as shown in Table 10.4.

As you can see, when the platform is powered up, only flash ROM is assigned a location in the memory map. The two SRAM banks (0 and 1) have not been initialized and are not available. The next stage is to bring in the two SRAM banks and remap flash ROM to a new location. This is achieved using the following code:

```
        LDR     r14, =sandstone_init2
        LDR     r4, =0x01800000         ; new flash ROM location
```

Table 10.4 Initial memory state.

Memory type	Start address	End address	Size
Flash ROM	0x00000000	0x00080000	512K
SRAM bank 0	Unavailable	unavailable	256K
SRAM bank 1	Unavailable	unavailable	256K

```
        ADD     r14, r14, r4
        ADRL    r0, memorymaptable_str
        LDMIA   r0, {r1-r12}
        LDR     r0, =EXTDBWTH           ; =(SYSCFG + 0x3010)
        STMIA   r0, {r1-r12}
        MOV     pc, r14                 ; jump to remapped memory

sandstone_init2
        ; Code after sandstone_init2 executes @ +0x1800000
```

The first part of the code calculates the absolute address of the routine *sandstone_init2* before remapping takes place. This address is used by Sandstone to jump to the next routine in the new remapped environment.

The second part carries out the memory remapping. The new memory map data is loaded into registers *r1* to *r12*, from a structure pointed by `memorymaptable_str`. This structure, using the registers, is then written to the memory controller offset `0x3010` from system configuration register. Once this is complete, the new memory map as shown in Table 10.5 is active.

You can see that the SRAM banks are now available, and the flash ROM is set to a higher address. The final part is to jump to the next routine, or stage, of the firmware.

This jump is achieved by taking advantage of the ARM pipeline. Even though the new memory environment is active, the next instruction has already been loaded into the pipeline. The next routine can be called by moving the contents of register *r14* (the address `sandstone_init2`) into the *pc*. We achieve this by using a single MOV instruction that follows immediately after the remap code.

Table 10.5 Remapping.

Type	Start address	End address	Size
Flash ROM	0x01800000	0x01880000	512K
SRAM bank 0	0x00000000	0x00040000	256K
SRAM bank 1	0x00040000	0x00080000	256K

The results of executing step 3 are the following:

- Memory has been remapped as shown in Table 10.5.
- *pc* now points to the next step. This address is located in the newly remapped flash ROM.

10.2.2.4 Step 4: Initialize Communication Hardware

Communication initialization involves configuring a serial port and outputting a standard banner. The banner is used to show that the firmware is fully functional and memory has been successfully remapped. Again, because the code for initializing the serial port on the ARM Evaluator-7T is hardware specific, it is not shown.

The serial port is set to 9600 baud, no parity, one stop bit, and no flow control. If a serial cable is attached to the board, then the host terminal has to be configured with these settings.

The results of executing step 4 are the following:

- Serial port initialized—9600 baud, no parity, one stop bit, and no flow control.
- Sandstone banner sent out through the serial port:

```
Sandstone Firmware (0.01)
- platform ......... e7t
- status ........... alive
- memory ........... remapped

+ booting payload ...
```

10.2.2.5 Step 5: Bootloader—Copy Payload and Relinquish Control

The final stage involves copying a payload and relinquishing control of the *pc* over to the copied payload. This is achieved using the code shown here. The first part of the code sets up the registers *r12*, *r13*, and *r14* used in the block copy. The bootloader code assumes that the payload is a plain binary image that requires no deciphering or uncompressing.

```
sandstone_load_and_boot
        MOV     r13,#0                    ; destination addr
        LDR     r12,payload_start_address  ; start addr
        LDR     r14,payload_end_address    ; end addr
```

```
_copy
        LDMIA   r12!,{r0-r11}
        STMIA   r13!,{r0-r11}
        CMP     r12,r14
        BLT     _copy
        MOV     pc,#0

payload_start_address
        DCD     startAddress
payload_end_address
        DCD     endAddress
```

Destination register *r13* points to the beginning of SRAM, in this case 0x00000000. The source register *r12* points to the start of the payload, and the source end register *r14* points to the end of the payload. Using these registers, the payload is then copied into SRAM.

Control of the *pc* is then relinquished to the payload by forcing the *pc* to the entry address of the copied payload. For this particular payload the entry point is address 0x00000000. The payload now has control of the system.

The results of executing step 5 are the following:

- Payload copied into SRAM, address 0x00000000.
- Control of the *pc* is relinquished to the payload; *pc* = 0x00000000.
- The system is completely booted. The following output is shown on the serial port:

```
Sandstone Firmware (0.01)
- platform ......... e7t
- status ........... alive
- memory ........... remapped

+ booting payload ...

Simple Little OS (0.09)
- initialized ...... ok
- running on ....... e7t

e7t:
```

CHAPTER 12
CACHES

A *cache* is a small, fast array of memory placed between the processor core and main memory that stores portions of recently referenced main memory. The processor uses cache memory instead of main memory whenever possible to increase system performance. The goal of a cache is to reduce the memory access bottleneck imposed on the processor core by slow memory.

Often used with a cache is a *write buffer*—a very small first-in-first-out (FIFO) memory placed between the processor core and main memory. The purpose of a write buffer is to free the processor core and cache memory from the slow write time associated with writing to main memory.

The word *cache* is a French word meaning "a concealed place for storage." When applied to ARM embedded systems, this definition is very accurate. The cache memory and write buffer hardware when added to a processor core are designed to be transparent to software code execution, and thus previously written software does not need to be rewritten for use on a cached core. Both the cache and write buffer have additional control hardware that automatically handles the movement of code and data between the processor and main memory. However, knowing the details of a processor's cache design can help you create programs that run faster on a specific ARM core.

Since the majority of this chapter is about the wonderful things a cache can do to make programs run faster, the question arises, "Are there any drawbacks created by having a cache in your system?" The answer is yes. The main drawback is the difficulty of determining the execution time of a program. Why this is a problem will become evident shortly.

Since cache memory only represents a very small portion of main memory, the cache fills quickly during program execution. Once full, the cache controller frequently evicts existing code or data from cache memory to make more room for the new code or data. This eviction process tends to occur randomly, leaving some data in cache and removing others. Thus, at any given instant in time, a value may or may not be stored in cache memory.

Because data may or may not be present in cache at any given point in time, the execution time of a routine may vary slightly from run to run due to the difference between the time it takes to use data immediately out of cache memory and the time it takes to load a cache line from main memory.

So, with that caveat, we begin by showing where caches fit in a standard memory hierarchy and introduce the principle of locality of reference to explain why a cache improves system performance. We then describe cache architectures in general and define a set of terms used by the ARM community. We end the chapter with example code showing how to clean and flush caches and to lock code and data segments in cache.

12.1 The Memory Hierarchy and Cache Memory

In Chapter 1 we introduced the memory hierarchy in a computer system. Figure 12.1 reviews some of this information to show where a cache and write buffer fit in the hierarchy.

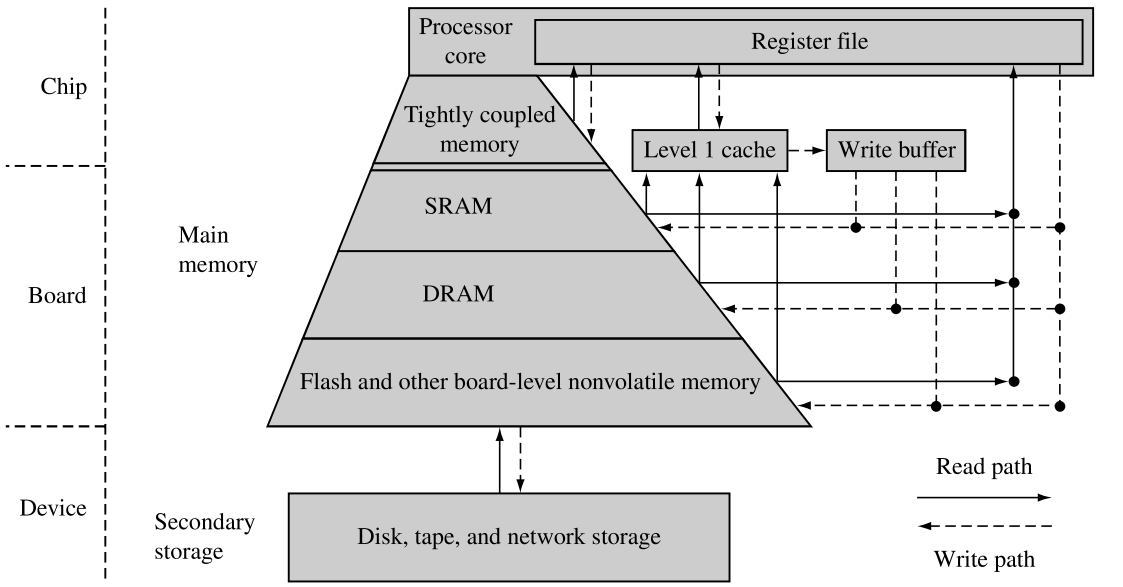


Figure 12.1 Memory hierarchy.

The innermost level of the hierarchy is at the processor core. This memory is so tightly coupled to the processor that in many ways it is difficult to think of it as separate from the processor. This memory is known as a *register file*. These registers are integral to the processor core and provide the fastest possible memory access in the system.

At the primary level, memory components are connected to the processor core through dedicated on-chip interfaces. It is at this level we find tightly coupled memory (TCM) and level 1 cache. We talk more about caches in a moment.

Also at the primary level is main memory. It includes volatile components like SRAM and DRAM, and nonvolatile components like flash memory. The purpose of main memory is to hold programs while they are running on a system.

The next level is secondary storage—large, slow, relatively inexpensive mass storage devices such as disk drives or removable memory. Also included in this level is data derived from peripheral devices, which are characterized by their extremely long access times. Secondary memory is used to store unused portions of very large programs that do not fit in main memory and programs that are not currently executing.

It is useful to note that a memory hierarchy depends as much on architectural design as on the technology surrounding it. For example, TCM and SRAM are of the same technology yet differ in architectural placement: TCM is located on the chip, while SRAM is located on a board.

A cache may be incorporated between any level in the hierarchy where there is a significant access time difference between memory components. A cache can improve system performance whenever such a difference exists. A cache memory system takes information stored in a lower level of the hierarchy and temporarily moves it to a higher level.

Figure 12.1 includes a level 1 (L1) cache and write buffer. The L1 cache is an array of high-speed, on-chip memory that temporarily holds code and data from a slower level. A cache holds this information to decrease the time required to access both instructions and data. The write buffer is a very small FIFO buffer that supports writes to main memory from the cache.

Not shown in the figure is a level 2 (L2) cache. An L2 cache is located between the L1 cache and slower memory. The L1 and L2 caches are also known as the *primary* and *secondary* caches.

Figure 12.2 shows the relationship that a cache has with main memory system and the processor core. The upper half of the figure shows a block diagram of a system without a cache. Main memory is accessed directly by the processor core using the datatypes supported by the processor core. The lower half of the diagram shows a system with a cache. The cache memory is much faster than main memory and thus responds quickly to data requests by the core. The cache's relationship with main memory involves the transfer of small blocks of data between the slower main memory to the faster cache memory. These blocks of data are known as cache lines. The write buffer acts as a temporary buffer that frees available space in the cache memory. The cache transfers a cache line to the write buffer at high speed and then the write buffer drains it to main memory at slow speed.

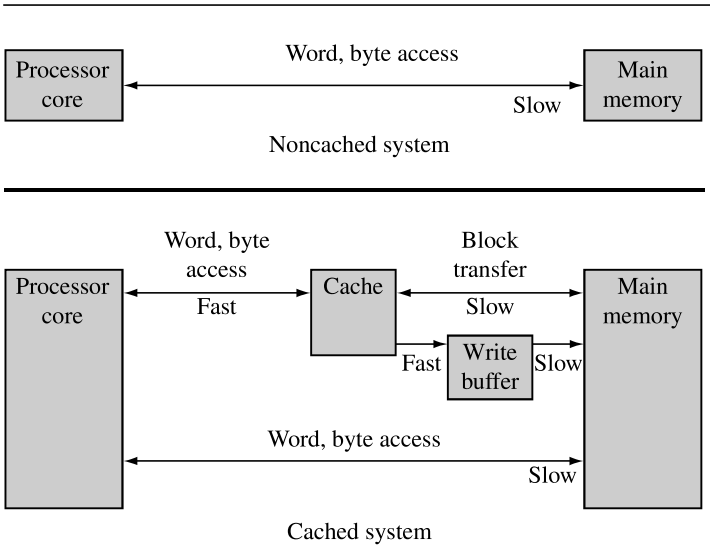


Figure 12.2 Relationship that a cache has between the processor core and main memory.

12.1.1 Caches and Memory Management Units

If a cached core supports virtual memory, it can be located between the core and the memory management unit (MMU), or between the MMU and physical memory. Placement of the cache before or after the MMU determines the addressing realm the cache operates in and how a programmer views the cache memory system. Figure 12.3 shows the difference between the two caches.

A *logical cache* stores data in a virtual address space. A logical cache is located between the processor and the MMU. The processor can access data from a logical cache directly without going through the MMU. A logical cache is also known as a *virtual cache.*

A *physical cache* stores memory using physical addresses. A physical cache is located between the MMU and main memory. For the processor to access memory, the MMU must first translate the virtual address to a physical address before the cache memory can provide data to the core.

ARM cached cores with an MMU use logical caches for processor families ARM7 through ARM10, including the Intel StrongARM and Intel XScale processors. The ARM11 processor family uses a physical cache. See Chapter 14 for additional information on the operation of the MMU.

The improvement a cache provides is possible because computer programs execute in nonrandom ways. Predictable program execution is the key to the success of cached systems. If a program's accesses to memory were random, a cache would provide little

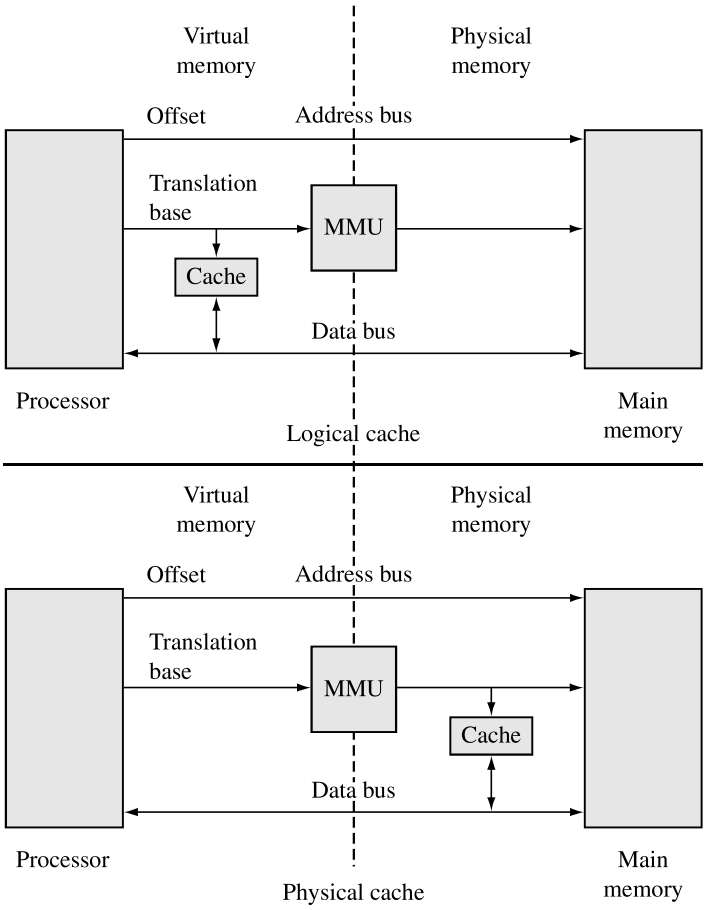


Figure 12.3 Logical and physical caches.

improvement to overall system performance. The principle of *locality of reference* explains the performance improvement provided by the addition of a cache memory to a system. This principle states that computer software programs frequently run small loops of code that repeatedly operate on local sections of data memory.

The repeated use of the same code or data in memory, or those very near, is the reason a cache improves performance. By loading the referenced code or data into faster memory when first accessed, each subsequent access will be much faster. It is the repeated access to the faster memory that improves performance.

The cache makes use of this repeated local reference in both time and space. If the reference is in time, it is called *temporal locality*. If it is by address proximity, then it is called *spatial locality*.

12.2 Cache Architecture

ARM uses two bus architectures in its cached cores, the Von Neumann and the Harvard. The Von Neumann and Harvard bus architectures differ in the separation of the instruction and data paths between the core and memory. A different cache design is used to support the two architectures.

In processor cores using the Von Neumann architecture, there is a single cache used for instruction and data. This type of cache is known as a *unified cache*. A unified cache memory contains both instruction and data values.

The Harvard architecture has separate instruction and data buses to improve overall system performance, but supporting the two buses requires two caches. In processor cores using the Harvard architecture, there are two caches: an instruction cache (I-cache) and a data cache (D-cache). This type of cache is known as a *split cache*. In a split cache, instructions are stored in the instruction cache and data values are stored in the data cache.

We introduce the basic architecture of caches by showing a unified cache in Figure 12.4. The two main elements of a cache are the cache *controller* and the cache *memory*. The cache memory is a dedicated memory array accessed in units called *cache lines*. The cache controller uses different portions of the address issued by the processor during a memory request to select parts of cache memory. We will present the architecture of the cache memory first and then proceed to the details of the cache controller.

12.2.1 Basic Architecture of a Cache Memory

A simple cache memory is shown on the right side of Figure 12.4. It has three main parts: a directory store, a data section, and status information. All three parts of the cache memory are present for each cache line.

The cache must know where the information stored in a cache line originates from in main memory. It uses a directory store to hold the address identifying where the cache line was copied from main memory. The directory entry is known as a *cache-tag*.

A cache memory must also store the data read from main memory. This information is held in the data section (see Figure 12.4).

The size of a cache is defined as the actual code or data the cache can store from main memory. Not included in the cache size is the cache memory required to support cache-tags or status bits.

There are also status bits in cache memory to maintain state information. Two common status bits are the valid bit and dirty bit. A *valid* bit marks a cache line as active, meaning it contains live data originally taken from main memory and is currently available to the

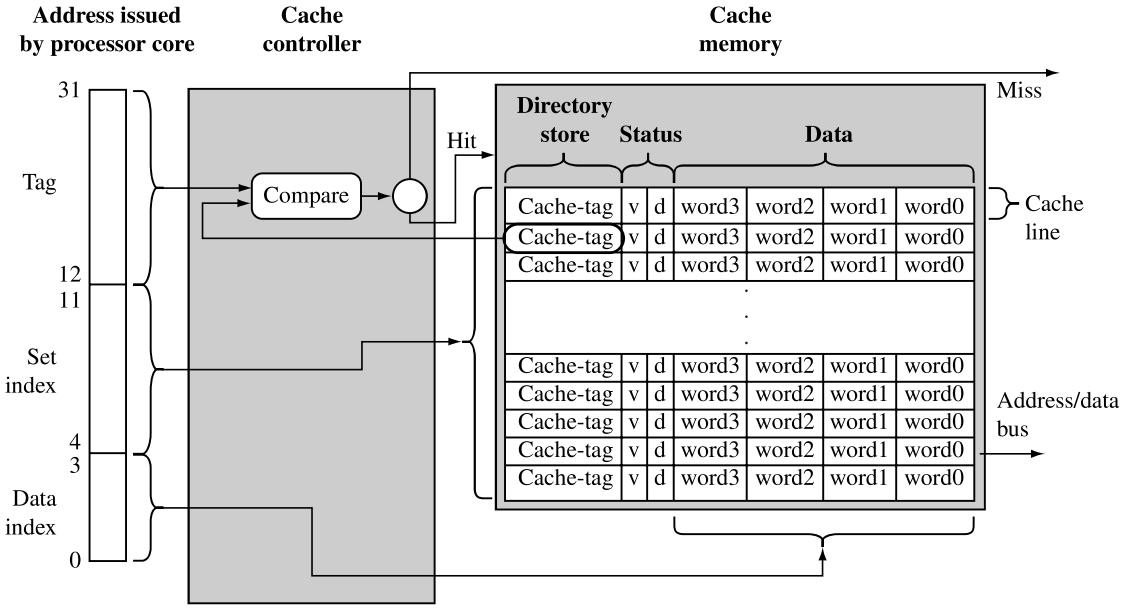


Figure 12.4 A 4 KB cache consisting of 256 cache lines of four 32-bit words.

processor core on demand. A *dirty* bit defines whether or not a cache line contains data that is different from the value it represents in main memory. We explain dirty bits in more detail in Section 12.3.1.

12.2.2 Basic Operation of a Cache Controller

The *cache controller* is hardware that copies code or data from main memory to cache memory automatically. It performs this task automatically to conceal cache operation from the software it supports. Thus, the same application software can run unaltered on systems with and without a cache.

The cache controller intercepts read and write memory requests before passing them on to the memory controller. It processes a request by dividing the address of the request into three fields, the *tag* field, the *set index* field, and the *data index* field. The three bit fields are shown in Figure 12.4.

First, the controller uses the set index portion of the address to locate the cache line within the cache memory that might hold the requested code or data. This cache line contains the cache-tag and status bits, which the controller uses to determine the actual data stored there.

The controller then checks the valid bit to determine if the cache line is active, and compares the cache-tag to the tag field of the requested address. If both the status check and comparison succeed, it is a cache *hit*. If either the status check or comparison fails, it is a cache *miss*.

On a cache miss, the controller copies an entire cache line from main memory to cache memory and provides the requested code or data to the processor. The copying of a cache line from main memory to cache memory is known as a *cache line fill*.

On a cache hit, the controller supplies the code or data directly from cache memory to the processor. To do this it moves to the next step, which is to use the data index field of the address request to select the actual code or data in the cache line and provide it to the processor.

12.2.3 The Relationship between Cache and Main Memory

Having a general understanding of basic cache memory architecture and how the cache controller works provides enough information to discuss the relationship that a cache has with main memory.

Figure 12.5 shows where portions of main memory are temporarily stored in cache memory. The figure represents the simplest form of cache, known as a *direct-mapped* cache. In a direct-mapped cache each addressed location in main memory maps to a single location in cache memory. Since main memory is much larger than cache memory, there are many addresses in main memory that map to the same single location in cache memory. The figure shows this relationship for the class of addresses ending in `0x824`.

The three bit fields introduced in Figure 12.4 are also shown in this figure. The set index selects the one location in cache where all values in memory with an ending address of `0x824` are stored. The data index selects the word/halfword/byte in the cache line, in this case the second word in the cache line. The tag field is the portion of the address that is compared to the cache-tag value found in the directory store. In this example there are one million possible locations in main memory for every one location in cache memory. Only one of the possible one million values in the main memory can exist in the cache memory at any given time. The comparison of the tag with the cache-tag determines whether the requested data is in cache or represents another of the million locations in main memory with an ending address of `0x824`.

During a cache line fill the cache controller may forward the loading data to the core at the same time it is copying it to cache; this is known as *data streaming*. Streaming allows a processor to continue execution while the cache controller fills the remaining words in the cache line.

If valid data exists in this cache line but represents another address block in main memory, the entire cache line is evicted and replaced by the cache line containing the requested address. This process of removing an existing cache line as part of servicing a cache miss is known as *eviction*—returning the contents of a cache line to main memory from the cache to make room for new data that needs to be loaded in cache.

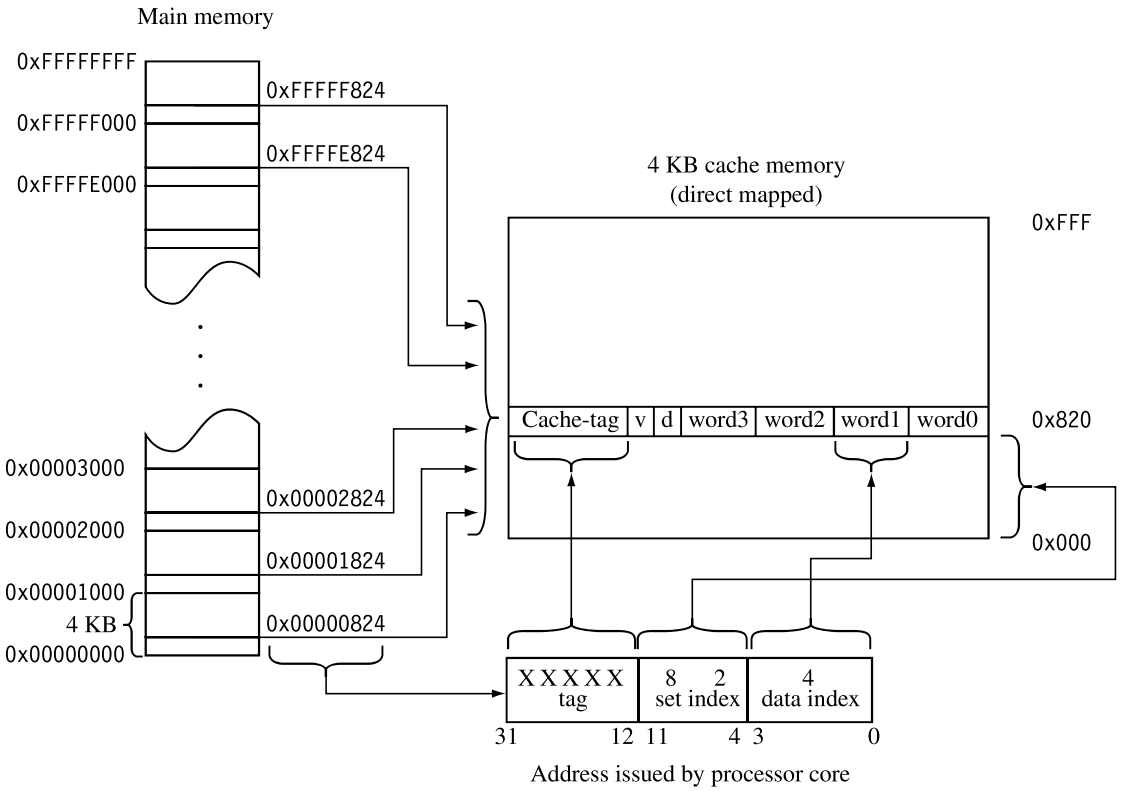


Figure 12.5 How main memory maps to a direct-mapped cache.

A direct-mapped cache is a simple solution, but there is a design cost inherent in having a single location available to store a value from main memory. Direct-mapped caches are subject to high levels of *thrashing*—a software battle for the same location in cache memory. The result of thrashing is the repeated loading and eviction of a cache line. The loading and eviction result from program elements being placed in main memory at addresses that map to the same cache line in cache memory.

Figure 12.6 takes Figure 12.5 and overlays a simple, contrived software procedure to demonstrate thrashing. The procedure calls two routines repeatedly in a `do while` loop. Each routine has the same set index address; that is, the routines are found at addresses in physical memory that map to the same location in cache memory. The first time through the loop, routine A is placed in the cache as it executes. When the procedure calls routine B, it evicts routine A a cache line at a time as it is loaded into cache and executed. On the second time through the loop, routine A replaces routine B, and then routine B replaces routine A.

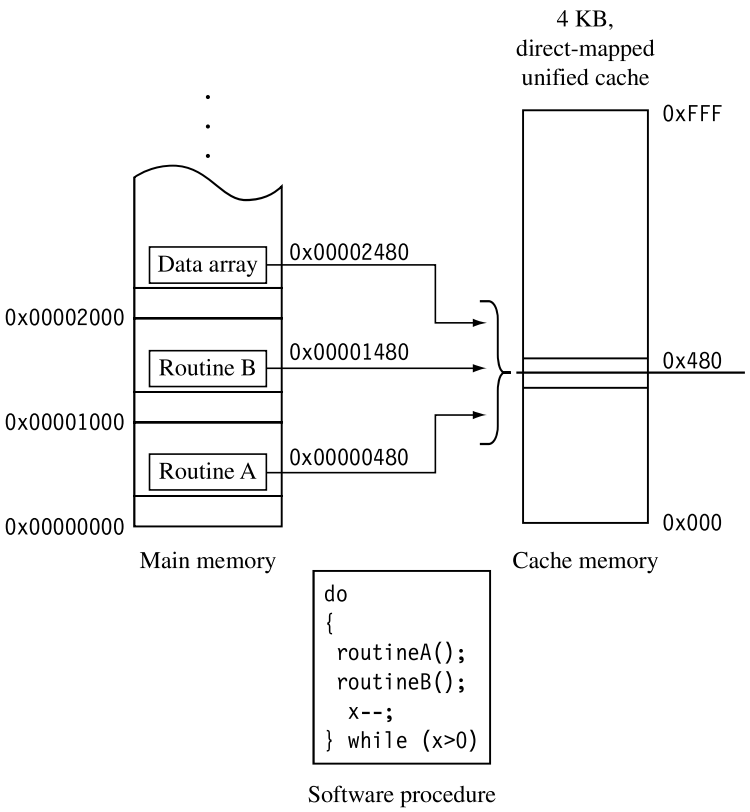


Figure 12.6 Thrashing: two functions replacing each other in a direct-mapped cache.

Repeated cache misses result in continuous eviction of the routine that not running. This is cache thrashing.

12.2.4 Set Associativity

Some caches include an additional design feature to reduce the frequency of thrashing (see Figure 12.7). This structural design feature is a change that divides the cache memory into smaller equal units, called *ways*. Figure 12.7 is still a four KB cache; however, the set index now addresses more than one cache line—it points to one cache line in each way. Instead of one way of 256 lines, the cache has four ways of 64 lines. The four cache lines with the same set index are said to be in the same *set*, which is the origin of the name "set index."

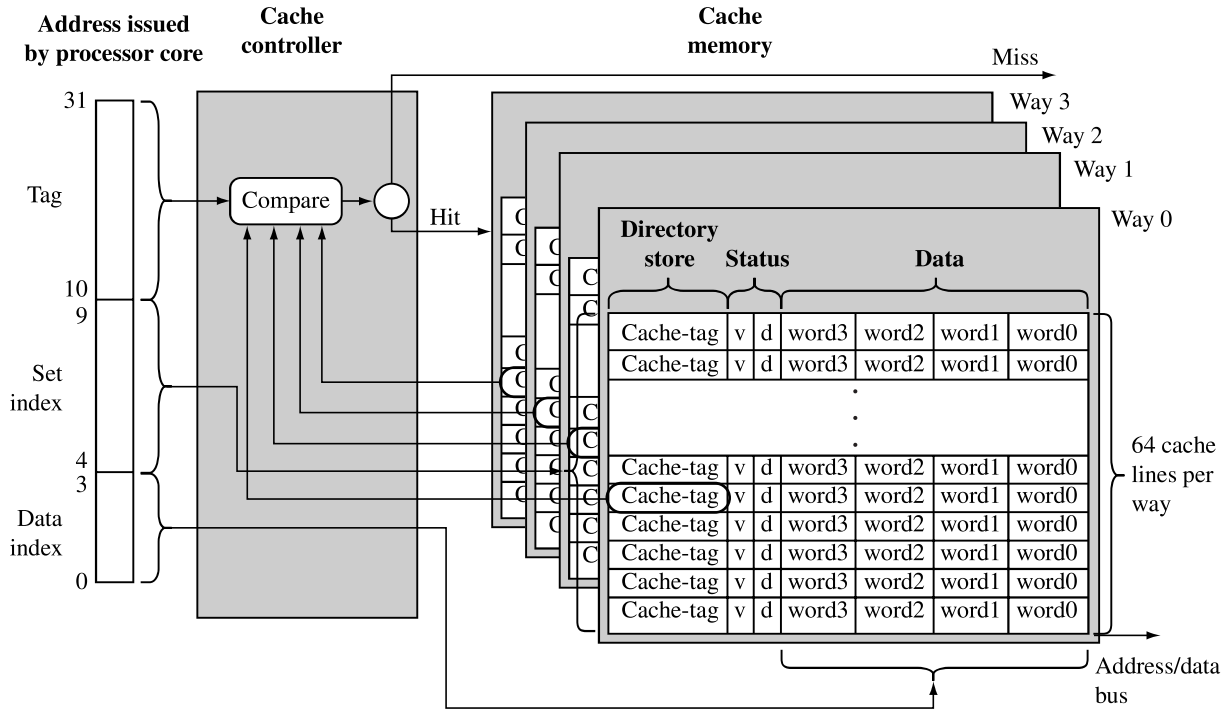


Figure 12.7 A 4 KB, four-way set associative cache. The cache has 256 total cache lines, which are separated into four ways, each containing 64 cache lines. The cache line contains four words.

The set of cache lines pointed to by the set index are *set associative*. A data or code block from main memory can be allocated to any of the four ways in a set without affecting program behavior; in other words the storing of data in cache lines within a set does not affect program execution. Two sequential blocks from main memory can be stored as cache lines in the same way or two different ways. The important thing to note is that the data or code blocks from a specific location in main memory can be stored in any cache line that is a member of a set. The placement of values within a set is exclusive to prevent the same code or data block from simultaneously occupying two cache lines in a set.

The mapping of main memory to a cache changes in a four-way set associative cache. Figure 12.8 shows the differences. Any single location in main memory now maps to four different locations in the cache. Although Figures 12.5 and 12.8 both illustrate 4 KB caches, here are some differences worth noting.

The bit field for the tag is now two bits larger, and the set index bit field is two bits smaller. This means four million main memory addresses now map to one set of four cache lines, instead of one million addresses mapping to one location.

The size of the area of main memory that maps to cache is now 1 KB instead of 4 KB. This means that the likelihood of mapping cache line data blocks to the same set is now four times higher. This is offset by the fact that a cache line is one fourth less likely to be evicted.

If the example code shown in Figure 12.6 were run in the four-way set associative cache shown in Figure 12.8, the incidence of thrashing would quickly settle down as routine A, routine B, and the data array would establish unique places in the four available locations in a set. This assumes that the size of each routine and the data are less than the new smaller 1 KB area that maps from main memory.

12.2.4.1 Increasing Set Associativity

As the associativity of a cache controller goes up, the probability of thrashing goes down. The ideal goal would be to maximize the set associativity of a cache by designing it so any main memory location maps to any cache line. A cache that does this is known as a *fully associative* cache. However, as the associativity increases, so does the complexity of the hardware that supports it. One method used by hardware designers to increase the set associativity of a cache includes a *content addressable memory* (CAM).

A CAM uses a set of comparators to compare the input tag address with a cache-tag stored in each valid cache line. A CAM works in the opposite way a RAM works. Where a RAM produces data when given an address value, a CAM produces an address if a given data value exists in the memory. Using a CAM allows many more cache-tags to be compared simultaneously, thereby increasing the number of cache lines that can be included in a set.

Using a CAM to locate cache-tags is the design choice ARM made in their ARM920T and ARM940T processor cores. The caches in the ARM920T and ARM940T are 64-way set associative. Figure 12.9 shows a block diagram of an ARM940T cache. The cache controller uses the address tag as the input to the CAM and the output selects the way containing the valid cache line.

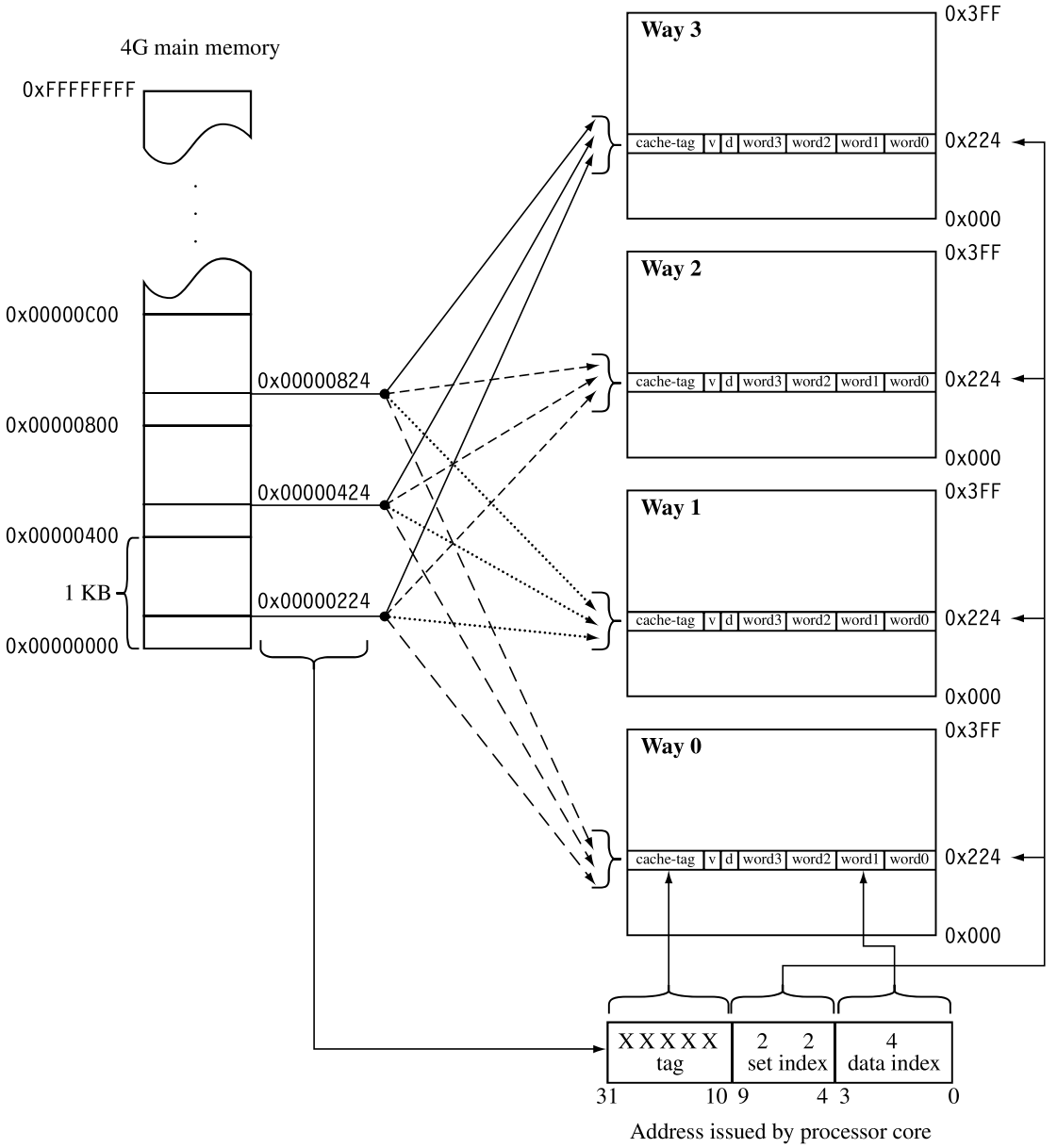


Figure 12.8 Main memory mapping to a four-way set associative cache.

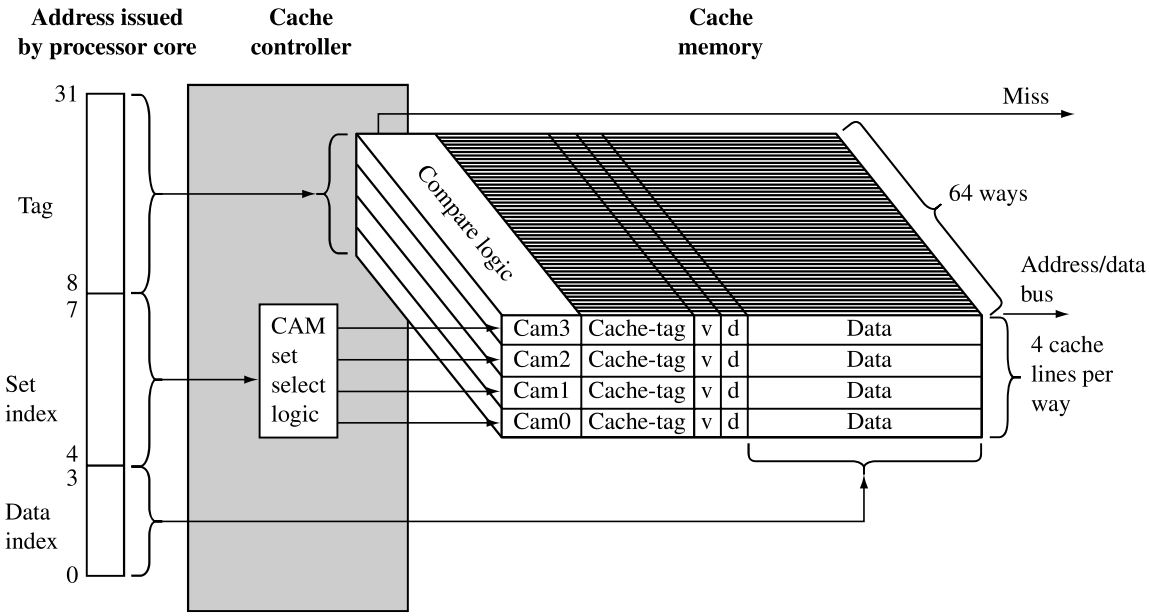


Figure 12.9 ARM940T—4 KB 64-way set associative D-cache using a CAM.

The tag portion of the requested address is used as an input to the four CAMs that simultaneously compare the input tag with all cache-tags stored in the 64 ways. If there is a match, cache data is provided by the cache memory. If no match occurs, a miss signal is generated by the memory controller.

The controller enables one of four CAMs using the set index bits. The indexed CAM then selects a cache line in cache memory and the data index portion of the core address selects the requested word, halfword, or byte within the cache line.

12.2.5 Write Buffers

A write buffer is a very small, fast FIFO memory buffer that temporarily holds data that the processor would normally write to main memory. In a system without a write buffer, the processor writes directly to main memory. In a system with a write buffer, data is written at high speed to the FIFO and then emptied to slower main memory. The write buffer reduces the processor time taken to write small blocks of sequential data to main memory. The FIFO memory of the write buffer is at the same level in the memory hierarchy as the L1 cache and is shown in Figure 12.1.

The efficiency of the write buffer depends on the ratio of main memory writes to the number of instructions executed. Over a given time interval, if the number of writes to main memory is low or sufficiently spaced between other processing instructions, the write buffer will rarely fill. If the write buffer does not fill, the running program continues to execute out of cache memory using registers for processing, cache memory for reads and writes, and the write buffer for holding evicted cache lines while they drain to main memory.

A write buffer also improves cache performance; the improvement occurs during cache line evictions. If the cache controller evicts a dirty cache line, it writes the cache line to the write buffer instead of main memory. Thus the new cache line data will be available sooner, and the processor can continue operating from cache memory.

Data written to the write buffer is not available for reading until it has exited the write buffer to main memory. The same holds true for an evicted cache line: it too cannot be read while it is in the write buffer. This is one of the reasons that the FIFO depth of a write buffer is usually quite small, only a few cache lines deep.

Some write buffers are not strictly FIFO buffers. The ARM10 family, for example, supports *coalescing*—the merging of write operations into a single cache line. The write buffer will merge the new value into an existing cache line in the write buffer if they represent the same data block in main memory. Coalescing is also known as *write merging*, *write collapsing*, or *write combining*.

12.2.6 Measuring Cache Efficiency

There are two terms used to characterize the cache efficiency of a program: the cache hit rate and the cache miss rate. The *hit rate* is the number of cache hits divided by the total number of memory requests over a given time interval. The value is expressed as a percentage:

$$hit\ rate = \left(\frac{cache\ hits}{memory\ requests} \right) \times 100$$

The *miss rate* is similar in form: the total cache misses divided by the total number of memory requests expressed as a percentage over a time interval. Note that the miss rate also equals 100 minus the hit rate.

The hit rate and miss rate can measure reads, writes, or both, which means that the terms can be used to describe performance information in several ways. For example, there is a hit rate for reads, a hit rate for writes, and other measures of hit and miss rates.

Two other terms used in cache performance measurement are the *hit time*—the time it takes to access a memory location in the cache and the *miss penalty*—the time it takes to load a cache line from main memory into cache.

12.3 Cache Policy

There are three policies that determine the operation of a cache: the write policy, the replacement policy, and the allocation policy. The cache write policy determines where data is stored during processor write operations. The replacement policy selects the cache line in a set that is used for the next line fill during a cache miss. The allocation policy determines when the cache controller allocates a cache line.

12.3.1 Write Policy—Writeback or Writethrough

When the processor core writes to memory, the cache controller has two alternatives for its write policy. The controller can write to both the cache and main memory, updating the values in both locations; this approach is known as *writethrough*. Alternatively, the cache controller can write to cache memory and not update main memory, this is known as *writeback* or *copyback*.

12.3.1.1 Writethrough

When the cache controller uses a writethrough policy, it writes to both cache and main memory when there is a cache hit on write, ensuring that the cache and main memory stay coherent at all times. Under this policy, the cache controller performs a write to main memory for each write to cache memory. Because of the write to main memory, a writethrough policy is slower than a writeback policy.

12.3.1.2 Writeback

When a cache controller uses a writeback policy, it writes to valid cache data memory and not to main memory. Consequently, valid cache lines and main memory may contain different data. The cache line holds the most recent data, and main memory contains older data, which has not been updated.

Caches configured as writeback caches must use one or more of the dirty bits in the cache line status information block. When a cache controller in writeback writes a value to cache memory, it sets the dirty bit true. If the core accesses the cache line at a later time, it knows by the state of the dirty bit that the cache line contains data not in main memory. If the cache controller evicts a dirty cache line, it is automatically written out to main memory. The controller does this to prevent the loss of vital information held in cache memory and not in main memory.

One performance advantage a writeback cache has over a writethrough cache is in the frequent use of temporary local variables by a subroutine. These variables are transient in nature and never really need to be written to main memory. An example of one of these

transient variables is a local variable that overflows onto a cached stack because there are not enough registers in the register file to hold the variable.

12.3.2 Cache Line Replacement Policies

On a cache miss, the cache controller must select a cache line from the available set in cache memory to store the new information from main memory. The cache line selected for replacement is known as a *victim*. If the victim contains valid, dirty data, the controller must write the dirty data from the cache memory to main memory before it copies new data into the victim cache line. The process of selecting and replacing a victim cache line is known as *eviction*.

The strategy implemented in a cache controller to select the next victim is called its *replacement policy*. The replacement policy selects a cache line from the available associative member set; that is, it selects the way to use in the next cache line replacement. To summarize the overall process, the set index selects the set of cache lines available in the ways, and the replacement policy selects the specific cache line from the set to replace.

ARM cached cores support two replacement policies, either pseudorandom or round-robin.

- Round-robin or cyclic replacement simply selects the next cache line in a set to replace. The selection algorithm uses a sequential, incrementing victim counter that increments each time the cache controller allocates a cache line. When the victim counter reaches a maximum value, it is reset to a defined base value.
- Pseudorandom replacement randomly selects the next cache line in a set to replace. The selection algorithm uses a nonsequential incrementing victim counter. In a pseudorandom replacement algorithm the controller increments the victim counter by randomly selecting an increment value and adding this value to the victim counter. When the victim counter reaches a maximum value, it is reset to a defined base value.

Most ARM cores support both policies (see Table 12.1 for a comprehensive list of ARM cores and the policies they support). The round-robin replacement policy has greater predictability, which is desirable in an embedded system. However, a round-robin replacement policy is subject to large changes in performance given small changes in memory access. To show this change in performance, we provide Example 12.1.

EXAMPLE 12.1 This example determines the time it takes to execute a software routine using the round-robin and random replacement policies. The test routine `cache_RRtest` collects timings using the clock function available in the C library header time.h. First, it enables a round robin policy and runs a timing test, and then enables the random policy and runs the same test.

The test routine `readSet` is written specifically for an ARM940T and intentionally shows a worst-case abrupt change in cache behavior using a round-robin replacement policy.

Table 12.1 ARM cached core policies.

Core	Write policy	Replacement policy	Allocation policy
ARM720T	writethrough	random	read-miss
ARM740T	writethrough	random	read-miss
ARM920T	writethrough, writeback	random, round-robin	read-miss
ARM940T	writethrough, writeback	random	read-miss
ARM926EJS	writethrough, writeback	random, round-robin	read-miss
ARM946E	writethrough, writeback	random, round-robin	read-miss
ARM10202E	writethrough, writeback	random, round-robin	read-miss
ARM1026EJS	writethrough, writeback	random, round-robin	read-miss
Intel StrongARM	writeback	round-robin	read-miss
Intel XScale	writethrough, writeback	round-robin	read-miss, write-miss

```
#include <stdio.h>
#include <time.h>

void cache_RRtest(int times,int numset)
{
   clock_t count;

   printf("Round Robin test size = %d\r\n", numset);
   enableRoundRobin();
   cleanFlushCache();
   count = clock();
   readSet(times,numset);
   count = clock() - count;
   printf("Round Robin enabled = %.2f seconds\r\n",
                                 (float)count/CLOCKS_PER_SEC);

   enableRandom();
   cleanFlushCache();
   count = clock();
   readSet(times, numset);
   count = clock() - count;
   printf("Random enabled = %.2f seconds\r\n\r\n",
                                 (float)count/CLOCKS_PER_SEC);
}

int readSet( int times, int numset)
{
```

```
  int setcount, value;
  volatile int *newstart;
  volatile int *start = (int *)0x20000;

  __asm
  {
    timesloop:
        MOV     newstart, start
        MOV     setcount, numset
    setloop:
        LDR     value,[newstart,#0];
        ADD     newstart,newstart,#0x40;
        SUBS    setcount, setcount, #1;
        BNE     setloop;
        SUBS    times, times, #1;
        BNE     timesloop;
  }
  return value;
}
```

We wrote the `readSet` routine to fill a single set in the cache. There are two arguments to the function. The first, `times`, is the number of times to run the test loop; this value increases the time it takes to run the test. The second, `numset`, is the number of set values to read; this value determines the number of cache lines the routine loads into the same set. Filling the set with values is done in a loop using an `LDR` instruction that reads a value from a memory location and then increments the address by 16 words (64 bytes) in each pass through the loop. Setting the value of `numset` to 64 will fill all the available cache lines in a set in an ARM940T. There are 16 words in a way and 64 cache lines per set in the ARM940T.

Here are two calls to the round-robin test using two set sizes. The first reads and fills a set with 64 entries; the second attempts to fill the set with 65 entries.

```
  unsigned int times = 0x10000;
  unsigned int numset = 64;

  cache_RRtest(times, numset);
  numset = 65;
  cache_RRtest(times, numset);
```

The console output of the two tests follows. The tests were run on an ARM940T core module simulated using the ARM ADS1.2 ARMulator with a core clock speed of 50 MHz and a memory read access time of 100 ns nonsequential and 50 ns sequential. The thing to notice is the change in timing for the round-robin test reading 65 set values.

```
Round Robin test size = 64
Round Robin enabled = 0.51 seconds
Random enabled = 0.51 seconds
Round Robin test size = 65
Round Robin enabled = 2.56 seconds
Random enabled = 0.58 seconds
```

This is an extreme example, but it does shows a difference between using a round-robin policy and a random replacement policy.

Another common replacement policy is *least recently used* (LRU). This policy keeps track of cache line use and selects the cache line that has been unused for the longest time as the next victim.

ARM's cached cores do not support a least recently used replacement policy, although ARM's semiconductor partners have taken noncached ARM cores and added their own cache to the chips they produce. So there are ARM-based products that use an LRU replacement policy.

12.3.3 Allocation Policy on a Cache Miss

There are two strategies ARM caches may use to allocate a cache line after a the occurrence of a cache miss. The first strategy is known as *read-allocate*, and the second strategy is known as *read-write-allocate*.

A read allocate on cache miss policy allocates a cache line only during a read from main memory. If the victim cache line contains valid data, then it is written to main memory before the cache line is filled with new data.

Under this strategy, a write of new data to memory does not update the contents of the cache memory unless a cache line was allocated on a previous read from main memory. If the cache line contains valid data, then a write updates the cache and may update main memory if the cache write policy is writethrough. If the data is not in cache, the controller writes to main memory only.

A read-write allocate on cache miss policy allocates a cache line for either a read or write to memory. Any load or store operation made to main memory, which is not in cache memory, allocates a cache line. On memory reads the controller uses a read-allocate policy.

On a write, the controller also allocates a cache line. If the victim cache line contains valid data, then it is first written back to main memory before the cache controller fills the victim cache line with new data from main memory. If the cache line is not valid, it simply does a cache line fill. After the cache line is filled from main memory, the controller writes the data to the corresponding data location within the cache line. The cached core also updates main memory if it is a writethrough cache.

The ARM7, ARM9, and ARM10 cores use a read-allocate on miss policy; the Intel XScale supports both read-allocate and write-allocate on miss. Table 12.1 provides a listing of the policies supported by each core.

12.4 Coprocessor 15 and Caches

There are several coprocessor 15 registers used to specifically configure and control ARM cached cores. Table 12.2 lists the coprocessor 15 registers that control cache configuration. Primary CP15 registers *c7* and *c9* control the setup and operation of cache. Secondary CP15:c7 registers are write only and clean and flush cache. The CP15:c9 register defines the victim pointer base address, which determines the number of lines of code or data that are locked in cache. We discuss these commands in more detail in the sections that follow. To review the general use of coprocessor 15 instructions and syntax, see Section 3.5.2.

There are other CP15 registers that affect cache operation; the definition of these registers is core dependent. These other registers are explained in Chapter 13 in Sections 13.2.3 and 13.2.4 on initializing the MPU, and in Chapter 14 in Section 14.3.6 on initializing the MMU.

In the next several sections we use the CP15 registers listed in Table 12.2 to provide example routines to clean and flush caches, and to lock code or data in cache. The control system usually calls these routines as part of its memory management activities.

12.5 Flushing and Cleaning Cache Memory

ARM uses the terms *flush* and *clean* to describe two basic operations performed on a cache.

To "flush a cache" is to clear it of any stored data. Flushing simply clears the valid bit in the affected cache line. All or just portions of a cache may need flushing to support changes in memory configuration. The term *invalidate* is sometimes used in place of the term *flush*. However, if some portion of the D-cache is configured to use a writeback policy, the data cache may also need cleaning.

To "clean a cache" is to force a write of dirty cache lines from the cache out to main memory and clear the dirty bits in the cache line. Cleaning a cache reestablishes coherence between cached memory and main memory, and only applies to D-caches using a writeback policy.

Table 12.2 Coprocessor 15 registers that configure and control cache operation.

Function	Primary register	Secondary registers	Opcode 2
Clean and flush cache	*c7*	*c5*, *c6*, *c7*, *c10*, *c13*, *c14*	0, 1, 2
Drain write buffer	*c7*	*c10*	4
Cache lockdown	*c9*	*c0*	0, 1
Round-robin replacement	*c15*	*c0*	0

K.S SCHOOL OF ENGINEERING AND MANAGEMENT
DEPARTMENT OF COMPUTER SCIENCE & ENGG.
Advanced Learners List

YEAR / SEMESTER - IV - 'A'			
COURSE TITLE - Microcontrollers			
COURSE CODE -BCS402			
S.NO	**USN**	**NAME**	**Signature**
1	1KG22CS002	A PAVITHRA	A. Pavithra
2	1KG22CS003	ABHISHEK S	AS. Abhishek
3	1KG22CS004	AKSHAYA K	Akshaya K
4	1KG22CS006	AMISHA V	Amisha. V
5	1KG22CS010	ANJANA R C	Anjana
6	1KG22CS022	C GOWTHAM	C Gowtham
7	1KG22CS033	DEEKSHA D SHENOY	Deeksha. D. Shenoy
8	1KG22CS040	GAGANA SHREE S	Gagana Shree. S
9	1KG22CS045	GORTHI YASWANTH	G. Yas
10	1KG22CS055	KAVANA S M	Kavana
11	1KG22CS056	KAVYA S	

FACULTY INCHARGE

HOD

K.S SCHOOL OF ENGINEERING AND MANAGEMENT
DEPARTMENT OF COMPUTER SCIENCE & ENGG.
Advanced Learners List

YEAR / SEMESTER - IV - 'A'			
COURSE TITLE - Microcontrollers			
COURSE CODE -BCS402			
S.NO	**USN**	**NAME**	**Signature**
1	1KG22CS002	A PAVITHRA	A. Pavithra.
2	1KG22CS003	ABHISHEK S	A. Abhishek
3	1KG22CS006	AMISHA V	Amisha.V
4	1KG22CS033	DEEKSHA D SHENOY	Deeksha. D. Shenoy
5	1KG22CS039	G UHA	G. Uha
6	1KG22CS040	GAGANA SHREE S	Gagana

FACULTY INCHARGE

HOD

K.S SCHOOL OF ENGINEERING AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

Attendance For Remedial Class

YEAR / SEMESTER	**IV - 'A'**
COURSE TITLE	**Microcontrollers**
COURSE CODE	**BCS402**
ACADEMIC YEAR	**2023-2024**

S.NO	USN	NAME	Students Signature
1	1KG22CS001	A G VISHNU	
2	1KG22CS008	ANANTHANENI KRISHNA SAI	
3	1KG22CS012	B M DARSHAN	
4	1KG22CS021	BYNI PURUSHOTHAM	
5	1KG22CS027	CHARAN KUMAR P K	
6	1KG22CS029	D MAHESH	
7	1KG22CS031	DANDA SHALINI	
8	1KG22CS034	DISHA S	
9	1KG22CS035	DIYA AJITH KASABEKAR	
10	1KG22CS036	E. MASHAN KUMAR	
11	1KG22CS047	HARSHA C V	
12	1KG22CS048	HARSHITHA C K	
13	1KG22CS051	JAJAPPAGARI SAI SREE	
14	1KG22CS052	JAMPULA ABHILASH	
15	1KG22CS053	K GAYATHRI	
16	1KG22CS059	KOUSIK N	
17	1KG22CS060	L R DHAYATRI	

SIGNATURE OF THE FACULTY

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S SCHOOL OF ENGINEERING AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

Attendance For Remedial Class

YEAR / SEMESTER	IV - 'A'
COURSE TITLE	Microcontrollers
COURSE CODE	BCS402
ACADEMIC YEAR	2023-2024

S.NO	USN	NAME	Students Signature
1	1KG22CS001	A G VISHNU	
2	1KG22CS008	ANANTHANENI KRISHNA SAI	
3	1KG22CS012	B M DARSHAN	
4	1KG22CS021	BYNI PURUSHOTHAM	
5	1KG22CS027	CHARAN KUMAR P K	
6	1KG22CS029	D MAHESH	
7	1KG22CS031	DANDA SHALINI	
8	1KG22CS034	DISHA S	
9	1KG22CS035	DIYA AJITH KASABEKAR	
10	1KG22CS036	E. MADHAN KUMAR	
11	1KG22CS048	HARSHITHA C K	
12	1KG22CS051	JAJAPPAGARI SAI SREE	
13	1KG22CS052	JAMPULA ABHILASH	
14	1KG22CS053	K GAYATHRI	
15	1KG22CS059	KOUSIK N	
16	1KG22CS060	L R DHAYATRI	

SIGNATURE OF THE FACULTY

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S SCHOOL OF ENGINEERING AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

IV A -Assignment Marks Sheet

YEAR / SEMESTER	IV - 'A'
COURSE TITLE	Microcontrollers
COURSE CODE	BCS402
ACADEMIC YEAR	2023-2024

S.NO	USN	NAME	Assignment 1 (25)	Assignment 2 (25)	Total	Total Reduced (20)	Student Signature
			10	10	20	10	
1	1KG22CS001	A G VISHNU					
2	1KG22CS002	A PAVITHRA	10	10	20	10	A. Pavithra.
3	1KG22CS003	ABHISHEK S	10	10	20	10	A. Abhishek
4	1KG22CS004	AKSHAYA K	10	10	20	10	Akshaya
5	1KG22CS005	ALLURU VENKATASAI JYOTHISHREDDY	10	10	20	10	A. Jyothish
6	1KG22CS006	AMISHA V	10	10	20	10	Amisha V
7	1KG22CS007	ANAGHA S	10	10	20	10	Anagha
8	1KG22CS008	ANANTHANENI KRISHNA SAI	10	10	20	10	Sai.
9	1KG22CS009	ANCHAL R S SINGH	10	10	20	10	Anchal
10	1KG22CS010	ANJANA R C	10	10	20	10	
11	1KG22CS011	AVINASH NAYAK M	10	10	20	10	Avinash
12	1KG22CS012	B M DARSHAN	10	10	20	10	B M Darshan
13	1KG22CS013	B N RUSHITHA	10	10	20	10	
14	1KG22CS014	B USHASREE	10	10	20	10	B. Usha
15	1KG22CS015	B V DEEKSHA JAIN	10	10	20	10	
16	1KG22CS016	BHANU PRIYA K	10	10	20	10	Bhanu
17	1KG22CS017	BHARATH R N	10	08	18	09	RN
18	1KG22CS018	BHAVYA D	10	10	20	10	
19	1KG22CS019	BHEEMANNA	10	10	20	10	
20	1KG22CS020	BOURISETTI CHAITANYA	10	10	20	10	Chaitanya
21	1KG22CS021	BYNI PURUSHOTHAM	10	10	20	10	Purushotham
22	1KG22CS022	C GOWTHAM	10	10	20	10	Gowtham
23	1KG22CS023	C R ANAGHA	10	10	20	10	ACR
24	1KG22CS024	CHAITHANYA C	10	10	20	10	
25	1KG22CS025	CHAITRA C	10	10	20	10	Chaitra
26	1KG22CS026	CHALLA PAVAN KUMAR	10	10	20	10	C Pavan
27	1KG22CS027	CHARAN KUMAR P K	10	10	20	10	
28	1KG22CS028	CHARAN T R	10	10	20	10	charan

S.NO	USN	NAME	Assignment 1 (25) 10	Assignment 2 (25) 10	Total 20	Total Reduced (20) 10	Student Signature
29	1KG22CS029	D MAHESH	10	10	20	10	
30	1KG22CS030	D SHREYAS	10	10	20	10	
31	1KG22CS031	DANDA SHALINI	10	10	20	10	
32	1KG22CS032	DASARI YASASWI NANDA	10	10	20	10	
33	1KG22CS033	DEEKSHA D SHENOY	10	10	20	10	
34	1KG22CS034	DISHA S	10	10	20	10	
35	1KG22CS035	DIYA AJITH KASABEKAR	10	10	20	10	
36	1KG22CS036	E. MASHAN KUMAR	10	10	20	10	
37	1KG22CS037	ENTURI LOKESH	10	10	20	10	
38	1KG22CS038	G SHARATH RAJ	10	10	20	10	
39	1KG22CS039	G UHA	10	10	20	10	
40	1KG22CS040	GAGANA SHREE S	10	10	20	10	
41	1KG22CS041	GANASHREE C N	10	10	20	10	
42	1KG22CS042	GOLLA KAVYA	10	10	20	10	
43	1KG22CS043	GOLLA KUSUMA	10	10	20	10	
44	1KG22CS044	GONUGUNTLA PRASHANTH KUMAR	10	10	20	10	
45	1KG22CS045	GORTHI YASWANTH	10	10	20	10	
46	1KG22CS046	HARISH R A	10	10	20	10	
47	1KG22CS047	HARSHA C V	10	10	20	10	
48	1KG22CS048	HARSHITHA C K	10	10	20	10	
49	1KG22CS049	HEMANTH R	10	10	20	10	
50	1KG22CS050	HRISHIKESH B S	10	10	20	10	
51	1KG22CS051	JAJAPPAGARI SAI SREE	10	10	20	10	
52	1KG22CS052	JAMPULA ABHILASH	10	10	20	10	
53	1KG22CS053	K GAYATHRI	10	10	20	10	
54	1KG22CS054	K PRAMOD KUMAR	10	10	20	10	
55	1KG22CS055	KAVANA S M	10	10	20	10	
56	1KG22CS056	KAVYA S	10	10	20	10	
57	1KG22CS057	KEERTHANA B	10	10	20	10	
58	1KG22CS058	KOLLA BHAVANA	10	10	20	10	
59	1KG22CS059	KOUSIK N	10	10	20	10	
60	1KG22CS060	L R DHAYATRI	10	10	20	10	
61	1KG22CS061	LIKHITHA P V	10	10	20	10	
62	1KG22CS062	LIKITHA R	10	10	20	10	
63	1KG23CS400	GOWTHAM T M	10	10	20	10	
64	1KG23CS401	IQRAA	10	08	18	9	
65	1KG23CS402	MUDDASSIR	10	08	18	9	
66	1KG22CS063	SAIJA.M	10	08	18	9	

SIGNATURE OF THE FACULTY

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S SCHOOL OF ENGINEERING AND MANAGEMENT

DEPARTMENT OF COMPUTER SCIENCE & ENGG.

YEAR / SEMESTER	**IV - 'A'**
COURSE TITLE	**Microcontrollers**
COURSE CODE	**BCS402**
ACADEMIC YEAR	**2023-2024**

S.NO	USN	NAME	IA1 (30)	IA2 (30)	IA3 (30)	Best of 2 AVG IA	Scale Down To (15)	ASS (10)	ASS (10)	AVG ASS (10)	IA + ASS (25)	Lab (25)	Grand Total (50)	Student Signature
1	1KG22CS001	A G VISHNU	0	4	16	10	5	8	8	8	13	19	32	
2	1KG22CS002	A PAVITHRA	27	30	AB	29	15	10	10	10	25	25	50	
3	1KG22CS003	ABHISHEK S	29	28	25	29	15	10	10	10	25	24	49	
4	1KG22CS004	AKSHAYA K	28	23	23	26	13	10	10	10	23	23	46	
5	1KG22CS005	ALLURU VENKATASAI JYOTHISHREDDY	21	10	13	17	9	10	10	10	19	23	42	
6	1KG22CS006	AMISHA V	29	29	AB	29	15	10	10	10	25	25	50	
7	1KG22CS007	ANAGHA S	15	8	18	17	9	10	10	10	19	23	42	
8	1KG22CS008	ANANTHANENI KRISHNA SAI	13	17	AB	15	8	10	10	10	18	23	41	
9	1KG22CS009	ANCHAL R S SINGH	22	9	17	20	10	10	10	10	20	23	43	
10	1KG22CS010	ANJANA R C	29	23	AB	26	13	9	9	9	22	24	46	
11	1KG22CS011	AVINASH NAYAK M	24	AB	22	23	12	10	10	10	22	24	46	
12	1KG22CS012	B M DARSHAN	8	7	10	9	5	10	10	10	15	21	36	
13	1KG22CS013	B N RUSHITHA	15	17	18	18	9	10	10	10	19	22	41	

S.NO	USN	NAME	IA1 (30)	IA2 (30)	IA3 (30)	Best of 2 AVG IA	Scale Down To (15)	ASS (10)	ASS (10)	AVG ASS (10)	IA + ASS (25)	Lab (25)	Grand Total (50)	Student Signature
14	1KG22CS014	B USHASREE	19	18	15	19	10	10	10	10	20	24	44	
15	1KG22CS015	B V DEEKSHA JAIN	18	11	5	15	8	10	10	10	18	23	41	
16	1KG22CS016	BHANU PRIYA K	24	18	22	23	12	10	10	10	22	23	45	
17	1KG22CS017	BHARATH R N	18	5	6	12	6	10	8	9	15	23	38	
18	1KG22CS018	BHAVYA D	17	19	AB	18	9	10	10	10	19	23	42	
19	1KG22CS019	BHEEMANNA	21	18	AB	20	10	10	10	10	20	23	43	
20	1KG22CS020	BOURISETTI CHAITANYA	22	21	AB	22	11	10	10	10	21	24	45	
21	1KG22CS021	BYNI PURUSHOTHAM	5	5	AB	5	3	10	10	10	13	23	36	
22	1KG22CS022	C GOWTHAM	27	15	19	23	12	10	10	10	22	23	45	
23	1KG22CS023	C R ANAGHA	18	AB	16	17	9	10	10	10	19	22	41	
24	1KG22CS024	CHAITHANYA C	16	0	12	14	7	10	10	10	17	19	36	
25	1KG22CS025	CHAITRA C	18	15	AB	17	9	10	10	10	19	23	42	
26	1KG22CS026	CHALLA PAVAN KUMAR	16	7	11	14	7	10	10	10	17	23	40	
27	1KG22CS027	CHARAN KUMAR P K	13	8	18	16	8	10	10	10	18	23	41	
28	1KG22CS028	CHARAN T R	21	15	20	21	11	10	10	10	21	24	45	
29	1KG22CS029	D MAHESH	13	10	12	13	7	10	10	10	17	23	40	
30	1KG22CS030	D SHREYAS	24	23	AB	24	12	10	10	10	22	24	46	

S.NO	USN	NAME	IA1 (30)	IA2 (30)	IA3 (30)	Best of 2 AVG IA	Scale Down To (15)	ASS (10)	ASS (10)	AVG ASS (10)	IA + ASS (25)	Lab (25)	Grand Total (50)	Student Signature
31	1KG22CS031	DANDA SHALINI	8	14	19	17	9	10	10	10	19	24	43	
32	1KG22CS032	DASARI YASASWI NANDA	18	AB	22	20	10	10	10	10	20	24	44	
33	1KG22CS033	DEEKSHA D SHENOY	29	29	AB	29	15	10	10	10	25	25	50	
34	1KG22CS034	DISHA S	11	6	10	11	6	10	10	10	16	24	40	
35	1KG22CS035	DIYA AJITH KASABEKAR	13	9	22	18	9	10	10	10	19	23	42	
36	1KG22CS036	E. MASHAN KUMAR	5	0	11	8	5	10	10	10	15	22	37	
37	1KG22CS037	ENTURI LOKESH	16	11	15	16	8	10	10	10	18	22	40	
38	1KG22CS038	G SHARATH RAJ	22	9	AB	16	8	10	10	10	18	24	42	
39	1KG22CS039	G UHA	24	25	AB	25	13	10	10	10	23	22	45	
40	1KG22CS040	GAGANA SHREE S	26	25	AB	26	13	10	10	10	23	24	47	
41	1KG22CS041	GANASHREE C N	23	16	AB	20	10	10	10	10	20	24	44	
42	1KG22CS042	GOLLA KAVYA	19	20	19	20	10	10	10	10	20	24	44	
43	1KG22CS043	GOLLA KUSUMA	23	11	15	19	10	10	10	10	20	23	43	
44	1KG22CS044	GONUGUNTLA PRASHANTH KUMAR	16	AB	18	17	9	10	10	10	19	24	43	
45	1KG22CS045	GORTHI YASWANTH	25	22	AB	24	12	10	10	10	22	24	46	
46	1KG22CS046	HARISH R A	21	21	AB	21	11	10	10	10	21	23	44	
47	1KG22CS047	HARSHA C V	AB	5	7	6	3	10	10	10	13	22	35	
48	1KG22CS048	HARSHITHA C K	10	6	13	12	6	10	10	10	16	21	37	

S.NO	USN	NAME	IA1 (30)	IA2 (30)	IA3 (30)	Best of 2 AVG IA	Scale Down To (15)	ASS (10)	ASS (10)	AVG ASS (10)	IA + ASS (25)	Lab (25)	Grand Total (50)	Student Signature
49	1KG22CS049	HEMANTH R	17	11	5	14	7	10	10	10	17	24	41	
50	1KG22CS050	HRISHIKESH B S	15	AB	17	16	8	10	10	10	18	22	40	
51	1KG22CS051	JAJAPPAGARI SAI SREE	12	6	10	11	6	10	10	10	16	21	37	
52	1KG22CS052	JAMPULA ABHILASH	11	2	5	8	5	10	10	10	15	20	35	
53	1KG22CS053	K GAYATHRI	7	4	AB	6	3	10	10	10	13	22	35	
54	1KG22CS054	K PRAMOD KUMAR	21	11	12	17	9	10	10	10	19	23	42	
55	1KG22CS055	KAVANA S M	26	24	AB	25	13	10	10	10	23	23	46	
56	1KG22CS056	KAVYA S	26	21	AB	24	12	10	10	10	22	24	46	
57	1KG22CS057	KEERTHANA B	24	24	AB	24	12	10	10	10	22	24	46	
58	1KG22CS058	KOLLA BHAVANA	23	19	AB	21	11	10	10	10	21	24	45	
59	1KG22CS059	KOUSIK N	14	8	9	12	6	10	10	10	16	23	39	
60	1KG22CS060	L R DHAYATRI	12	4	17	15	8	10	10	10	18	24	42	
61	1KG22CS061	LIKHITHA P V	17	12	17	17	9	10	10	10	19	23	42	
62	1KG22CS062	LIKITHA R	15	12	15	15	8	10	10	10	18	22	40	
63	1KG23CS400	Gowtham T M	16	15	9	16	8	10	8	9	17	22	39	
64	1KG23CS401	Iqraa	19	12	AB	16	8	10	10	10	18	24	42	
65	1KG23CS402	Muddassir	16	AB	15	16	8	10	8	9	17	21	38	

SIGNATURE OF THE FACULTY

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S.SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MICROCONTROLLERS LABORATORY

IV Semester Lab Internals B-Form - JUL - 2024

BATCH: - A1 **SUBJECT CODE: - BCS402** **DATE: - 30/07/2024**

SEMESTER: - IV **SECTION: - A** **TIME: - 09.30 AM To 12.30 PM**

S.No	USN	Name	Signature	Batch No	Time
1	1KG22CS001	A G VISHNU		BATCH A1	09.30 AM To 12.30 PM
2	1KG22CS002	A PAVITHRA			
3	1KG22CS003	ABHISHEK S			
4	1KG22CS004	AKSHAYA K			
5	1KG22CS005	ALLURU VENKATASAI JYOTHISHREDDY			
6	1KG22CS006	AMISHA V			
7	1KG22CS007	ANAGHA S			
8	1KG22CS008	ANANTHANENI KRISHNA SAI			
9	1KG22CS009	ANCHAL R S SINGH			
10	1KG22CS010	ANJANA R C			
11	1KG22CS011	AVINASH NAYAK M			
12	1KG22CS012	B M DARSHAN			
13	1KG22CS013	B N RUSHITHA			
14	1KG22CS014	B USHASREE			
15	1KG22CS015	B V DEEKSHA JAIN			
16	1KG22CS016	BHANU PRIYA K			
17	1KG22CS017	BHARATH R N			
18	1KG22CS018	BHAVYA D			
19	1KG22CS019	BHEEMANNA			
20	1KG22CS020	BOURISETTI CHAITANYA			
21	1KG22CS021	BYNI PURUSHOTHAM			
22	1KG22CS022	C GOWTHAM			
23	1KG22CS023	C R ANAGHA			
24	1KG22CS024	CHAITHANYA C			
25	1KG22CS025	CHAITRA C			
26	1KG22CS026	CHALLA PAVAN KUMAR			
27	1KG22CS027	CHARAN KUMAR P K			
28	1KG22CS028	CHARAN T R			
29	1KG22CS029	D MAHESH			
30	1KG22CS030	D SHREYAS			
31	1KG22CS031	DANDA SHALINI			
32	1KG22CS032	DASARI YASASWI NANDA			
33	1KG22CS033	DEEKSHA D SHENOY			

Mrs. Meena G

30/7/24

Mrs. Sushmitha Suresh

Faculty Signature

HOD

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S.SCHOOL OF ENGINEERING AND MANAGEMENT, BENGALURU-560109

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

MICROCONTROLLERS LABORATORY

IV Semester Lab Internals B-Form - JUL - 2024

BATCH: - A2 **SUBJECT CODE: - BCS402** **DATE: - 29/07/2024**

SEMESTER: - IV **SECTION: - A** **TIME: - 09.30 AM To 12.30 PM**

S.No	USN	Name	Signature	Batch No	Time
1	1KG22CS034	DISHA S			
2	1KG22CS035	DIYA AJITH KASABEKAR			
3	1KG22CS036	E MADAN KUMAR			
4	1KG22CS037	ENTURI LOKESH			
5	1KG22CS038	G SHARATH RAJ			
6	1KG22CS039	G UHA			
7	1KG22CS040	GAGANA SHREE S			
8	1KG22CS041	GANASHREE C N			
9	1KG22CS042	GOLLA KAVYA			
10	1KG22CS043	GOLLA KUSUMA			
11	1KG22CS044	GONUGUNTLA PRASHANTH KUMAR			
12	1KG22CS045	GORTHI YASWANTH			
13	1KG22CS046	HARISH R A			
14	1KG22CS047	HARSHA C V			
15	1KG22CS048	HARSHITHA C K			
16	1KG22CS049	HEMANTH R		BATCH A2	09.30 AM To 12.30 PM
17	1KG22CS050	HRISHIKESH B S			
18	1KG22CS051	JAJAPPAGARI SAI SREE			
19	1KG22CS052	JAMPULA ABHILASH			
20	1KG22CS053	K GAYATHRI			
21	1KG22CS054	K PRAMOD KUMAR			
22	1KG22CS055	KAVANA S M			
23	1KG22CS056	KAVYA S			
24	1KG22CS057	KEERTHANA B			
25	1KG22CS058	KOLLA BHAVANA			
26	1KG22CS059	KOUSIK N			
27	1KG22CS060	L R DHAYATRI			
28	1KG22CS061	LIKHITHA P V			
29	1KG22CS062	LIKITHA R			
30	1KG22CS063	M SAIJA			
31	1KG23CS400	GOWTHAM TM			
32	1KG23CS401	IQRAA IMAN KHAN			
33	1KG23CS402	MUDDASSIR AHMED I TORGUL			

Mrs. Meena G

29/7/24

Mrs. Sushmitha Suresh

Faculty Signature

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT- 560 109

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Session 2024-2025 (EVEN Semester)

MICROCONTROLLERS LABORATORY

BATCH: - A1 SUBJECT CODE: - BCS402 DATE: - 30/07/2024

SEMESTER: - IV SECTION: - A TIME: - 09.30 AM To 12.30 PM

S.No	USN	Record	Observation	Out of 15	Write up	Execution	Result	Viva	Total 50	Out of 10	Total 25	Signature
		10	5		10	20	10	10				
1	1KG22CS001	9	4	13	7	10	10	1	28	6	19	
2	1KG22CS002	10	5	15	10	20	10	8	48	10	25	A. Pavithra.
3	1KG22CS003	10	5	15	10	20	10	5	45	9	24	K. Abhishek
4	1KG22CS004	10	5	15	10	20	10	2	42	8	23	
5	1KG22CS005	10	5	15	10	20	10	2	42	8	23	
6	1KG22CS006	10	5	15	10	20	10	8	48	10	25	Amisha. V
7	1KG22CS007	10	5	15	10	18	10	4	42	8	23	
8	1KG22CS008	10	5	15	9	19	10	2	40	8	23	
9	1KG22CS009	10	5	15	10	20	10	2	42	8	23	
10	1KG22CS010	10	5	15	10	20	10	5	45	9	24	
11	1KG22CS011	10	5	15	10	20	10	4	44	9	24	
12	1KG22CS012	10	5	15	0	20	7	1	28	6	21	
13	1KG22CS013	10	5	15	0	20	10	3	33	7	22	
14	1KG22CS014	10	5	15	10	20	10	4	44	9	24	Usha.
15	1KG22CS015	10	5	15	8	18	10	2	38	8	23	
16	1KG22CS016	10	5	15	10	20	10	0	40	8	23	
17	1KG22CS017	10	5	15	9	18	10	2	39	8	23	
18	1KG22CS018	10	5	15	10	20	4	4	38	8	23	
19	1KG22CS019	10	5	15	10	20	10	1	41	8	23	
20	1KG22CS020	10	5	15	10	20	10	4	44	9	24	
21	1KG22CS021	10	5	15	10	20	10	0	40	8	23	
22	1KG22CS022	10	5	15	10	20	10	2	42	8	23	
23	1KG22CS023	10	5	15	0	20	10	3	33	7	22	
24	1KG22CS024	8	5	13	7	10	10	1	28	6	19	
25	1KG22CS025	10	5	15	9	20	8	2	39	8	23	
26	1KG22CS026	10	5	15	9	20	7	2	38	8	23	
27	1KG22CS027	10	5	15	10	20	10	2	42	8	23	
28	1KG22CS028	10	5	15	10	20	10	3	43	9	24	
29	1KG22CS029	10	5	15	10	20	10	1	41	8	23	
30	1KG22CS030	10	5	15	10	20	10	7	47	9	24	
31	1KG22CS031	10	5	15	10	20	10	7	47	9	24	Shalini
32	1KG22CS032	10	5	15	10	20	10	3	43	9	24	
33	1KG22CS033	10	5	15	10	20	10	8	48	10	25	

Mrs. Meena G

30/7/24

Mrs. Sushmitha Suresh

Faculty Signature

HOD

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K. S. SCHOOL OF ENGINEERING AND MANAGEMENT- 560 109

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

Session 2024-2025 (EVEN Semester)

MICROCONTROLLERS LABORATORY

BATCH: - A2 SUBJECT CODE: - BCS402 DATE: - 29/07/2024

SEMESTER: - IV SECTION: - A TIME: - 09.30 AM To 12.30 PM

S.No	USN	Record	Observation	Out of 15	Write up	Execution	Result	Viva	Total 50	Out of 10	Total	Signature
		10	5		10	20	10	10				
1	1KG22CS034	10	5	15	10	20	10	4	44	9	24	
2	1KG22CS035	10	5	15	10	20	10	2	42	8	23	
3	1KG22CS036	10	5	15	8	20	7	1	36	7	22	
4	1KG22CS037	10	5	15	0	20	10	4	34	7	22	
5	1KG22CS038	10	5	15	10	20	10	4	44	9	24	
6	1KG22CS039	10	5	15	10	10	10	4	34	7	22	
7	1KG22CS040	10	5	15	10	20	10	4	44	9	24	
8	1KG22CS041	10	5	15	10	20	10	3	43	9	24	
9	1KG22CS042	10	5	15	10	20	10	4	44	9	24	
10	1KG22CS043	10	5	15	10	20	10	2	42	8	23	
11	1KG22CS044	10	5	15	10	20	10	3	43	9	24	
12	1KG22CS045	10	5	15	10	20	10	3	43	9	24	
13	1KG22CS046	10	5	15	10	20	10	2	42	8	23	
14	1KG22CS047	10	5	15	9	15	10	1	35	7	22	
15	1KG22CS048	10	5	15	0	20	8	4	32	6	21	
16	1KG22CS049	10	5	15	10	20	10	4	44	9	24	
17	1KG22CS050	10	5	15	0	20	10	5	35	7	22	
18	1KG22CS051	10	5	15	0	20	8	4	32	6	21	
19	1KG22CS052	10	5	15	0	15	10	1	26	5	20	
20	1KG22CS053	10	5	15	10	15	10	0	35	7	22	
21	1KG22CS054	10	5	15	10	20	10	2	42	8	23	
22	1KG22CS055	10	5	15	10	20	10	2	42	8	23	
23	1KG22CS056	10	5	15	10	20	10	4	44	9	24	
24	1KG22CS057	10	5	15	10	20	10	5	45	9	24	
25	1KG22CS058	10	5	15	10	20	10	5	45	9	24	
26	1KG22CS059	10	5	15	10	20	10	2	42	8	23	
27	1KG22CS060	10	5	15	10	20	10	4	44	9	24	
28	1KG22CS061	10	5	15	10	20	10	1	41	8	23	
29	1KG22CS062	10	5	15	10	15	10	1	36	7	22	
30	1KG22CS063	10	5	15	9	10	10	4	33	7	22	
31	1KG23CS400	10	5	15	10	20	10	4	44	9	24	
32	1KG23CS401	8	5	13	10	20	10	1	41	8	21	
33	1KG23CS402	8	5	13	9	20	8	2	39	8	21	

Mrs. Meena G 29/7/24

Mrs. Sushmitha Suresh 30/7/24

Faculty Signature

HOD

Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE

Branch : CS Semester : 4

Sl NO.	USN	BCS402
1	1KG21CS020	19 (**TH**) , 23 (**PR**)
2	1KG22CS001	13 (**TH**) , 19 (**PR**)
3	1KG22CS002	25 (**TH**) , 25 (**PR**)
4	1KG22CS003	25 (**TH**) , 24 (**PR**)
5	1KG22CS004	23 (**TH**) , 23 (**PR**)
6	1KG22CS005	19 (**TH**) , 23 (**PR**)
7	1KG22CS006	25 (**TH**) , 25 (**PR**)
8	1KG22CS007	19 (**TH**) , 23 (**PR**)
9	1KG22CS008	18 (**TH**) , 23 (**PR**)
10	1KG22CS009	20 (**TH**) , 23 (**PR**)
11	1KG22CS010	22 (**TH**) , 24 (**PR**)
12	1KG22CS011	22 (**TH**) , 24 (**PR**)
13	1KG22CS012	15 (**TH**) , 21 (**PR**)
14	1KG22CS013	19 (**TH**) , 22 (**PR**)
15	1KG22CS014	20 (**TH**) , 24 (**PR**)
16	1KG22CS015	18 (**TH**) , 23 (**PR**)
17	1KG22CS016	22 (**TH**) , 23 (**PR**)
18	1KG22CS017	15 (**TH**) , 23 (**PR**)
19	1KG22CS018	19 (**TH**) , 23 (**PR**)
20	1KG22CS019	20 (**TH**) , 23 (**PR**)
21	1KG22CS020	21 (**TH**) , 24 (**PR**)
22	1KG22CS021	13 (**TH**) , 23 (**PR**)
23	1KG22CS022	22 (**TH**) , 23 (**PR**)
24	1KG22CS023	19 (**TH**) , 22 (**PR**)
25	1KG22CS024	17 (**TH**) , 19 (**PR**)
26	1KG22CS025	19 (**TH**) , 23 (**PR**)
27	1KG22CS026	17 (**TH**) , 23 (**PR**)
28	1KG22CS027	18 (**TH**) , 23 (**PR**)
29	1KG22CS028	21 (**TH**) , 24 (**PR**)
30	1KG22CS029	17 (**TH**) , 23 (**PR**)
31	1KG22CS030	22 (**TH**) , 24 (**PR**)
32	1KG22CS031	19 (**TH**) , 24 (**PR**)
33	1KG22CS032	20 (**TH**) , 24 (**PR**)
34	1KG22CS033	25 (**TH**) , 25 (**PR**)
35	1KG22CS034	16 (**TH**) , 24 (**PR**)
36	1KG22CS035	19 (**TH**) , 23 (**PR**)

Sl NO.	USN	BCS402
37	1KG22CS036	15 (**TH**) , 22 (**PR**)
38	1KG22CS037	18 (**TH**) , 22 (**PR**)
39	1KG22CS038	18 (**TH**) , 24 (**PR**)
40	1KG22CS039	23 (**TH**) , 22 (**PR**)
41	1KG22CS040	23 (**TH**) , 24 (**PR**)
42	1KG22CS041	20 (**TH**) , 24 (**PR**)
43	1KG22CS042	20 (**TH**) , 24 (**PR**)
44	1KG22CS043	20 (**TH**) , 23 (**PR**)
45	1KG22CS044	19 (**TH**) , 24 (**PR**)
46	1KG22CS045	22 (**TH**) , 24 (**PR**)
47	1KG22CS046	21 (**TH**) , 23 (**PR**)
48	1KG22CS047	13 (**TH**) , 22 (**PR**)
49	1KG22CS048	16 (**TH**) , 21 (**PR**)
50	1KG22CS049	17 (**TH**) , 24 (**PR**)
51	1KG22CS050	18 (**TH**) , 22 (**PR**)
52	1KG22CS051	16 (**TH**) , 21 (**PR**)
53	1KG22CS052	15 (**TH**) , 20 (**PR**)
54	1KG22CS053	13 (**TH**) , 22 (**PR**)
55	1KG22CS054	19 (**TH**) , 23 (**PR**)
56	1KG22CS055	23 (**TH**) , 23 (**PR**)
57	1KG22CS056	22 (**TH**) , 24 (**PR**)
58	1KG22CS057	22 (**TH**) , 24 (**PR**)
59	1KG22CS058	21 (**TH**) , 24 (**PR**)
60	1KG22CS059	16 (**TH**) , 23 (**PR**)
61	1KG22CS060	18 (**TH**) , 24 (**PR**)
62	1KG22CS061	19 (**TH**) , 23 (**PR**)
63	1KG22CS062	18 (**TH**) , 22 (**PR**)
64	1KG22CS063	17 (**TH**) , 22 (**PR**)
65	1KG22CS064	21 (**TH**) , 24 (**PR**)
66	1KG22CS065	21 (**TH**) , 23 (**PR**)
67	1KG22CS066	12 (**TH**) , 23 (**PR**)
68	1KG22CS067	21 (**TH**) , 24 (**PR**)
69	1KG22CS068	16 (**TH**) , 21 (**PR**)
70	1KG22CS069	16 (**TH**) , 23 (**PR**)
71	1KG22CS070	19 (**TH**) , 23 (**PR**)
72	1KG22CS071	22 (**TH**) , 23 (**PR**)
73	1KG22CS072	16 (**TH**) , 21 (**PR**)
74	1KG22CS073	16 (**TH**) , 21 (**PR**)
75	1KG22CS074	18 (**TH**) , 23 (**PR**)

Sl NO.	USN	BCS402
76	1KG22CS075	16 **(TH)** , 21 **(PR)**
77	1KG22CS076	15 **(TH)** , 22 **(PR)**
78	1KG22CS077	20 **(TH)** , 23 **(PR)**
79	1KG22CS078	23 **(TH)** , 25 **(PR)**
80	1KG22CS079	17 **(TH)** , 21 **(PR)**
81	1KG22CS080	22 **(TH)** , 24 **(PR)**
82	1KG22CS081	14 **(TH)** , 23 **(PR)**
83	1KG22CS082	17 **(TH)** , 21 **(PR)**
84	1KG22CS083	23 **(TH)** , 25 **(PR)**
85	1KG22CS084	19 **(TH)** , 21 **(PR)**
86	1KG22CS085	20 **(TH)** , 23 **(PR)**
87	1KG22CS086	21 **(TH)** , 23 **(PR)**
88	1KG22CS087	20 **(TH)** , 24 **(PR)**
89	1KG22CS088	17 **(TH)** , 20 **(PR)**
90	1KG22CS089	12 **(TH)** , 16 **(PR)**
91	1KG22CS090	20 **(TH)** , 23 **(PR)**
92	1KG22CS091	18 **(TH)** , 22 **(PR)**
93	1KG22CS092	17 **(TH)** , 23 **(PR)**
94	1KG22CS093	21 **(TH)** , 24 **(PR)**
95	1KG22CS094	24 **(TH)** , 24 **(PR)**
96	1KG22CS095	22 **(TH)** , 25 **(PR)**
97	1KG22CS096	24 **(TH)** , 25 **(PR)**
98	1KG22CS097	19 **(TH)** , 22 **(PR)**
99	1KG22CS098	21 **(TH)** , 22 **(PR)**
100	1KG22CS099	16 **(TH)** , 22 **(PR)**
101	1KG22CS100	21 **(TH)** , 25 **(PR)**
102	1KG22CS101	16 **(TH)** , 20 **(PR)**
103	1KG22CS102	22 **(TH)** , 25 **(PR)**
104	1KG22CS103	18 **(TH)** , 23 **(PR)**
105	1KG22CS104	21 **(TH)** , 24 **(PR)**
106	1KG22CS105	18 **(TH)** , 24 **(PR)**
107	1KG22CS106	13 **(TH)** , 21 **(PR)**
108	1KG22CS107	15 **(TH)** , 20 **(PR)**
109	1KG22CS108	17 **(TH)** , 23 **(PR)**
110	1KG22CS109	14 **(TH)** , 14 **(PR)**
111	1KG22CS110	17 **(TH)** , 18 **(PR)**
112	1KG22CS111	18 **(TH)** , 23 **(PR)**
113	1KG22CS112	22 **(TH)** , 24 **(PR)**
114	1KG22CS113	15 **(TH)** , 23 **(PR)**

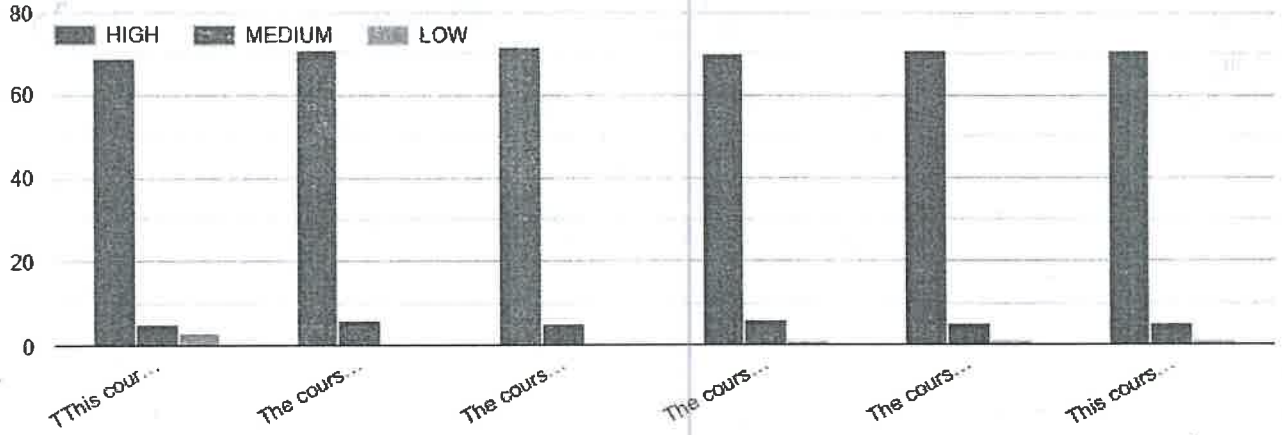
Sl NO.	USN	BCS402
115	1KG22CS114	19 (**TH**) , 22 (**PR**)
116	1KG22CS115	23 (**TH**) , 24 (**PR**)
117	1KG22CS116	16 (**TH**) , 22 (**PR**)
118	1KG22CS117	19 (**TH**) , 23 (**PR**)
119	1KG22CS118	15 (**TH**) , 22 (**PR**)
120	1KG22CS119	20 (**TH**) , 23 (**PR**)
121	1KG22CS120	19 (**TH**) , 20 (**PR**)
122	1KG22CS121	17 (**TH**) , 24 (**PR**)
123	1KG22CS122	21 (**TH**) , 24 (**PR**)
124	1KG22CS123	21 (**TH**) , 24 (**PR**)
125	1KG22CS124	19 (**TH**) , 23 (**PR**)
126	1KG22CS125	12 (**TH**) , 18 (**PR**)
127	1KG23CS400	18 (**TH**) , 24 (**PR**)
128	1KG23CS401	17 (**TH**) , 21 (**PR**)
129	1KG23CS402	17 (**TH**) , 21 (**PR**)
130	1KG23CS403	12 (**TH**) , 19 (**PR**)
131	1KG23CS404	14 (**TH**) , 19 (**PR**)
132	1KG23CS405	14 (**TH**) , 14 (**PR**)
133	1KG23CS406	12 (**TH**) , 18 (**PR**)

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COURSE END SURVEY -IV 'A'

Timestamp	Email	Name	USN	The course increased your level of interest? [TThis course has increased your level of interest on Microcontrollers..]	The course increased your level of interest? [The course content was understandable and was presented in a structured manner.]	The course increased your level of interest? [The course assignments/tests asses what I have learned in this course.]	The course increased your level of interest? [The course improved my ability to develop assembly language programs for ARM7-LPC2148 processor.]	The course increased your level of interest? [The course has given enough knowledge about Embedded system design.]	The course increased your level of interest? [This course has given you enough understanding to take up next level courses.]	Student's Signature
7/30/2024 12:03:51	Vishnuappilli@gmail.com	AG Vishnu	1KG22CS001	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 12:23:40	pavithra1512004@gmail.com	A Pavithra	1KG22CS002	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:42:56	s.abhishek140205@gmail.com	Abhishek S	1KG22CS003	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:43:44	akshaya.krishna.4k@gmail.com	Akshaya K	1KG22CS004	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 15:57:07	allurujyothish2004@gmail.com	Alluru Venkata Sai Jyothish Reddy	1KG22CS005	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:52:15	amishavamisha@gmail.com	Amisha.V	1KG22CS006	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/30/2024 12:05:51	anaghas2020@gmail.com	Anagha S	1KG22CS007	LOW	HIGH	HIGH	MEDIUM	HIGH	HIGH	
7/26/2024 14:33:50	krishnasai8782@gmail.com	A.Krishna Sai	1KG22CS008	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 15:51:45	anchalrssingh@gmail.com	Anchal	1KG22CS009	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 16:33:07	anjanarc1305@gmail.com	Anjana R.C	1KG22CS010	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 15:56:24	avvvviiiii11@gmail.com	Avinash Nayak	1KG22CS011	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 17:13:23	bangidarshan3@email.com	B M Darshan	1KG22CS012	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:49:17	rushithabnreddy@gmail.com	BN Rushitha	1KG22CS013	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:33:43	bushasree04@gmail.com	B.Usha Sree	1KG22CS014	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 11:10:49	bvdeekshajain19@gmail.com	B V DEEKSHA JAIN	1KG22CS015	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 11:00:56	kgowdabhanupriya@gmail.com	Bhanupriya K	1KG22CS016	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:33:52	bharath.rn.2004@gamil.com	BHARATH R N	1KG22CS017	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/28/2024 17:13:57	bhavyadnaidu47@gmail.com	Bhavya D	1KG22CS018	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 11:41:44	bchaitanya7174@gmail.com	BOURISETTI CHAITANYA	1KG22CS020	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 13:02:18	bynipurushotham04@gmail.com	Byni purushotham	1KG22CS021	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 19:12:13	gowthamnaidu979@gmail.com	Gowtham C	1KG22CS022	HIGH	HIGH	HIGH	MEDIUM	MEDIUM	MEDIUM	
7/27/2024 17:00:03	Anaghacr@gmail.com	CR anagha	1KG22CS023	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	
7/26/2024 14:38:48	Chaithanya84313@gmail.com	Chaithanya c	1KG22cs024	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:45:12	chaitranayaka203@gmail.com	Chaitra c	1KG22CS025	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:32:36	challapavan456@gmail.com	Challa Pavan Kumar	1KG22CS026	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 14:38:14	charannadiu05@gmail.com	Charan kumar pk	1KG22CS027	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 14:41:15	charantr123@gmail.com	Charan T R	1KG22CS028	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:32:38	dm701627@gmail.com	D Mahesh	1KG22CS029	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/30/2024 11:45:37	shreyasdsd1@gmail.com	D.shreyas	1KG22CS030	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:54:45	dandashalini448@gmail.com	D shalini	1KG22CS031	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:35:33	yasaswinanda@gmail.com	Dasari Yasaswi Nanda	1KG22CS032	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 12:39:15	deekshadsheroy24@gmail.com	Deeksha D Shenoy	1KG22CS033	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:31:03	dishasatish2004@gmail.com	Disha S	1KG22CS034	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	
7/30/2024 14:25:25	-	diya	1KG22CS035	LOW	MEDIUM	MEDIUM	LOW	LOW	MEDIUM	

K.S. SCHOOL OF ENGINEERING AND MANAGEMENT, BANGALORE - 560109
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COURSE END SURVEY -IV 'A'

Timestamp	Email	Name	USN	The course increased your level of interest? [TThis course has increased your level of interest on Microcontrollers..]	The course increased your level of interest? [The course content was understandable and was presented in a structured manner.]	The course increased your level of interest? [The course assignments/tests asses what I have learned in this course.]	The course increased your level of interest? [The course improved my ability to develop assembly language programs for ARM7-LPC2148 processor.]	The course increased your level of interest? [The course has given enough knowledge about Embedded system design.]	The course increased your level of interest? [This course has given you enough understanding to take up next level courses.]	Student's Signature
7/28/2024 13:10:39	madhanellanti@gmail.com	E.Madhan kumar	1KG22CS036	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:39:09	lokilokesh4031@gmail.com	E.LOKESH	1KG22CS037	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:32:33	rajsharath323@gmail.com	G sharath raj	1KG22CS038	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 12:44:30	gudetiuha@gmail.com	G Uha	1KG22CS039	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 11:02:17	gaganashrees26@gmail.com	Gagana Shree S	1KG22CS040	MEDIUM	HIGH	HIGH	MEDIUM	HIGH	HIGH	
7/27/2024 16:38:08	cnganashree@gmail.com	Ganashree C N	1KG22CS041	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 17:10:05	kavyagnavs25@gmail.com	G Kavya	1KG22CS042	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:39:00	gollakusumayadav@gmail.com	Golla Kusuma	1KG22CS043	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:35:32	gonuguntlaprashanth6301@gmai	Gonuguntla Prashanth Kumar	1KG22CS044	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 15:59:01	gorthiyaswanth60@gmail.com	G Yaswanth	1KG22CS045	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 17:42:51	harishra2500@gmail.com	Harish R A	1KG22CS046	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/30/2024 14:17:08	harshacv2004@gmail.com	Harsha	1KG22CS047	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 14:38:48	harshithack3210@gmail.com	Harshitha C K	1KG22CS048	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 19:25:35	hemanthbubby007@gmail.com	Hemanth R	1KG22CS049	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 15:48:57	hrishi2186@gmail.com	Hrishikesh B S	1KG22CS050	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 12:22:13	jajappagarisaisree2266@gmail.co	J.saisree	1KG22CS051	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 15:50:21	jampulaabhilash086@gmail.com	Jampula Abhilash	1KG22CS052	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/30/2024 14:49:27	ngayathri929@gmail.com	K.Gayathri	1KG22CS053	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 22:02:45	kpramodkuruba1@gmail.com	K Pramod Kumar	1KG22CS054	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/28/2024 17:15:30	kavana9980@gmail.com	Kavana S M	1KG22CS055	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 17:08:23	kavyabullappa@gmail.com	Kavya S	1KG22CS056	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:57:07	Keerthanagby@gmail.com	Keerthana B	1KG22CS057	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:32:54	kollabhavana2728@gmail.com	K. Bhavana	1KG22CS058	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 16:35:10	kollabhavana2728@gmail.com	K. Bhavana	1KG22CS058	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 10:33:32	koushikkoushikkousi@gmail.com	KOUSHIK	1KG22CS059	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 14:38:20	Lekkalapudidhayatri@gmail.com	Dhayatri	1KG22CS060	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 16:39:18	likhitha83107@gmail.com	Likhitha P V	1KG22CS061	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/27/2024 16:01:32	Likitha02vj@gmail.com	Likitha R	1KG22CS062	LOW	MEDIUM	MEDIUM	HIGH	MEDIUM	LOW	
7/27/2024 9:17:48	msaija53@gmail.com	Saija	1KG22CS063	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	MEDIUM	
7/26/2024 10:37:45	gowthamchoya5@gmail.com	Gowtham	1KG23CS400	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 14:23:58	iqraaimank@gmail.com	Iqraa Iman khan	1KG23CS401	HIGH	HIGH	HIGH	HIGH	HIGH	HIGH	
7/26/2024 11:05:20	imtorgal786@gmail.com	Muddassir Ahmed I Torgal	1KG23CS402	MEDIUM	HIGH	HIGH	HIGH	HIGH	HIGH	



FACULTY SIGNATURE

HOD

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109

K. S. School of Engineering & Management, Bangalore - 560109

Department of Computer Science Engineering

Staff Feedback (2023-24) Even Sem

Fourth Sem 'A' Section

Faculty Name: Mrs. Meena G

Course Name & Code: Micro controllers/BCS402

Class Strength:67

Sl. No.	1. Effective Planning & organisation	2. Punctuality / Class time Utilization	3. Ability to teach /explain / effective use of board	4. Interaction / Motivating students	5. Subject knowledge	6. Presentation of the subject / communication	7. Linking subject with practical application	8. Syllabus coverage / exam point of view	9. Evaluation of test / counselling	10. Attitude towards students
1	5	5	5	5	5	5	5	5	5	5
2	5	5	5	5	5	5	5	5	5	5
3	5	5	5	5	5	5	5	5	5	5
4	5	5	5	5	5	5	5	5	5	5
5	5	5	5	5	5	5	5	5	5	5
6	5	5	5	5	5	5	5	5	5	5
7	5	5	5	5	5	5	5	5	5	5
8	5	5	5	5	5	5	5	5	5	5
9	4	4	4	4	4	4	4	4	4	3
10	5	5	5	4	5	5	4	5	5	4
11	5	5	5	5	5	5	5	5	5	5
12	5	5	5	5	5	5	5	5	5	5
13	5	5	5	5	5	5	5	5	5	5
14	5	5	5	5	5	5	5	5	5	5
15	4	4	4	4	4	4	4	4	4	4
16	5	5	4	5	5	5	5	5	5	5
17	5	5	5	5	5	5	5	5	5	5
18	5	5	5	5	5	5	5	5	5	5
19	5	5	5	5	5	5	5	5	5	5
20	5	5	5	5	5	5	5	5	5	5
21	5	5	5	4	5	5	5	5	5	4
22	5	5	5	5	4	5	5	4	4	5
23	5	5	5	5	5	5	5	5	5	5
24	5	5	5	5	5	5	5	5	5	5
25	5	5	5	5	5	5	5	5	5	5
26	5	5	5	5	5	5	5	5	5	5
27	5	5	5	5	5	5	5	5	5	5
28	5	5	5	5	5	5	5	5	5	5
29	5	5	5	5	5	5	5	5	5	5
Col. Total	143	143	142	141	142	143	142	142	142	140
Col. Avg.	4.93	4.93	4.90	4.86	4.90	4.93	4.90	4.90	4.90	4.83
Over all %	97.93									

Head of Department

Principal

HOD
Department of Computer Science Engineering
K.S School of Engineering & Management
Bangalore-560109