MICROPROCESSORS AND MICROCONTROLLERS

Dr. P. Lachi Reddy

Professor in ECE

LBRCE, Mylavaram

UNIT-III

ARM ARCHITECTURE & PROGRAMMING MODEL

TOPICS - 1

- History
- ARM Features
- ARM Design Philosophy
- Registers
- Program Status Register
- Instruction Pipeline
- Interrupts and Vector Table

TOPICS -2

- ARM Processor Families
- Addressing Modes
- Instruction Set
- Data Processing Instructions
- Branch, Load - Store Instructions
- PSR Instructions and
- Conditional Instructions

History

- Developed the first ARM Processor (Acorn RISC Machine) in 1985 at Acorn Computers Limited.
- Established a new company named Advanced RISC Machine Limited and developed ARM6.
- Continuation of the architecture enhancements from the original architecture
- The ARM processor core is key component of many successful 32-bit embedded systems.

ARM Features

- A large register file
- A load/store architecture
- Uniform and fixed length instruction field
- Simple addressing mode
- Arithmetic Logic Unit and barrel shifter
- Auto increment and decrement addressing mode
- Conditional execution of instructions

Architecture Basics

- ARM cores use a 32-bit, Load-Store RISC architecture.
- That means that the core cannot directly manipulate the memory.
- All data manipulation must be done by loading registers with information located in memory, performing the data operation and then storing the value back to memory.
- There are 37 total registers in the processor.
- 37 Registers are split among seven different processor modes.

Architecture Basics -2

- The seven processor modes are used to run user tasks, an operating system, and to efficiently handle exceptions such as interrupts.
- Some of the registers with in each mode are reserved for specific use by the core, while most are available for general use.
- r13 is commonly used as the stack pointer (SP),
- r14 as a link register (LR)
- r15as a program counter (PC)
- The Current Program Status Register (CPSR), and the Saved Program Status Register (SPSR).

RISC Design Philosophy

- The design philosophy aimed at delivering the following.
  - simple but powerful instructions
  - single cycle execution at a high clock speed
  - intelligence in software rather than hardware
  - Provide greater flexibility on reducing the complexity of instructions.

- The RISC philosophy is implemented with four major design rules:
- Instructions: Reduced no. of instructions
- Pipelines: Parallel execution by pipeline
- Registers: Large general purpose register set
- Load-Store Architecture: Separate Load and Store instructions

ARM Design Philosophy

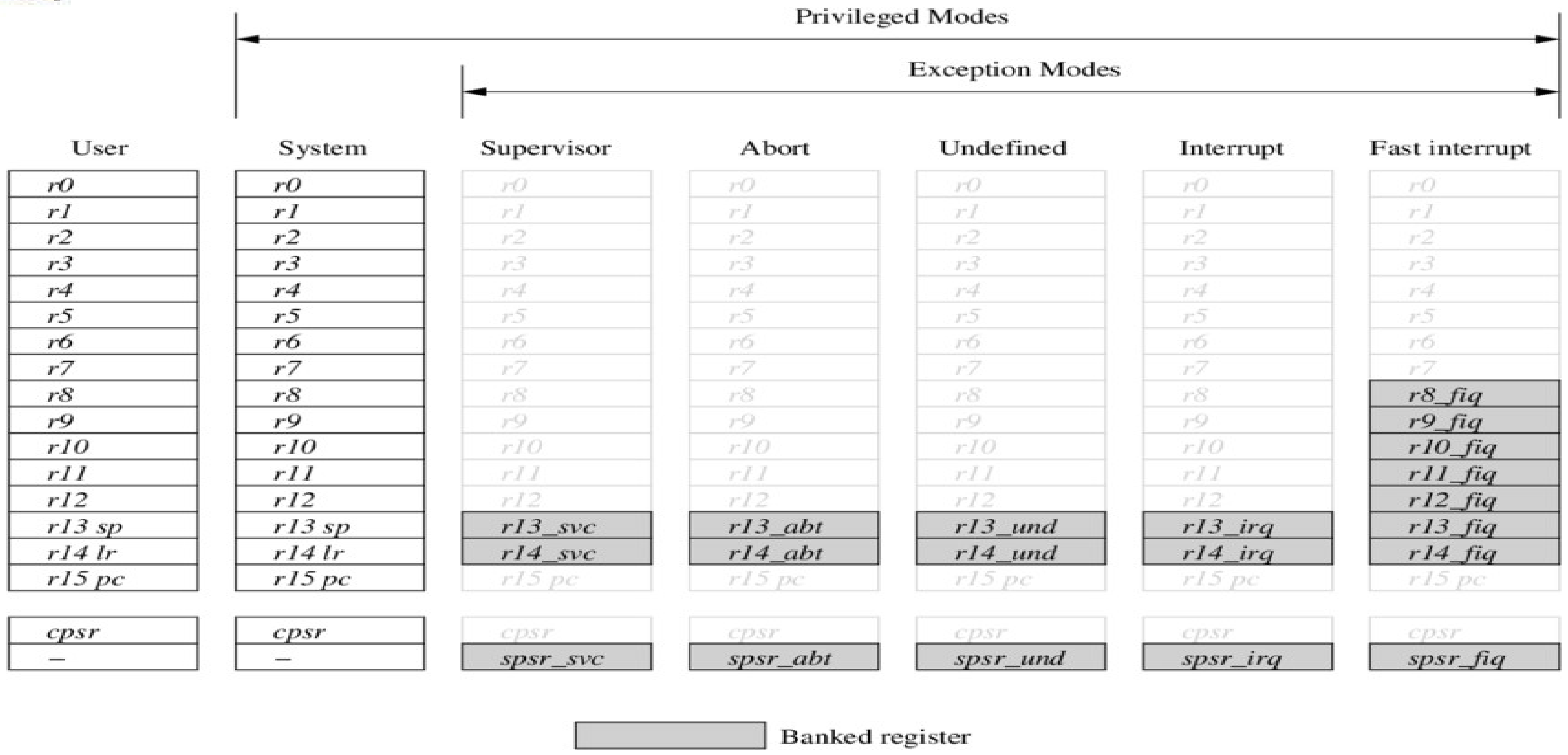
- There are a number of physical features that have driven the ARM processor design.

1. Small to reduce power consumption and extend battery operation
2. High code density
3. Price sensitive and use slow and low-cost memory devices.
4. Reduce the area of the die taken up by the embedded processor.
5. Hardware debug technology
6. ARM core is not a pure RISC architecture

Registers

- ARM processors provide general-purpose and special-purpose registers. Some additional registers are available in privileged execution modes.
- In all ARM processors, the following registers are available and accessible in any processor mode:
  - 13 general-purpose registers R0-R12.
  - One *Stack Pointer* (SP).
  - One *Link Register* (LR).
  - One *Program Counter* (PC).
  - One *Application Program Status Register* (APSR).

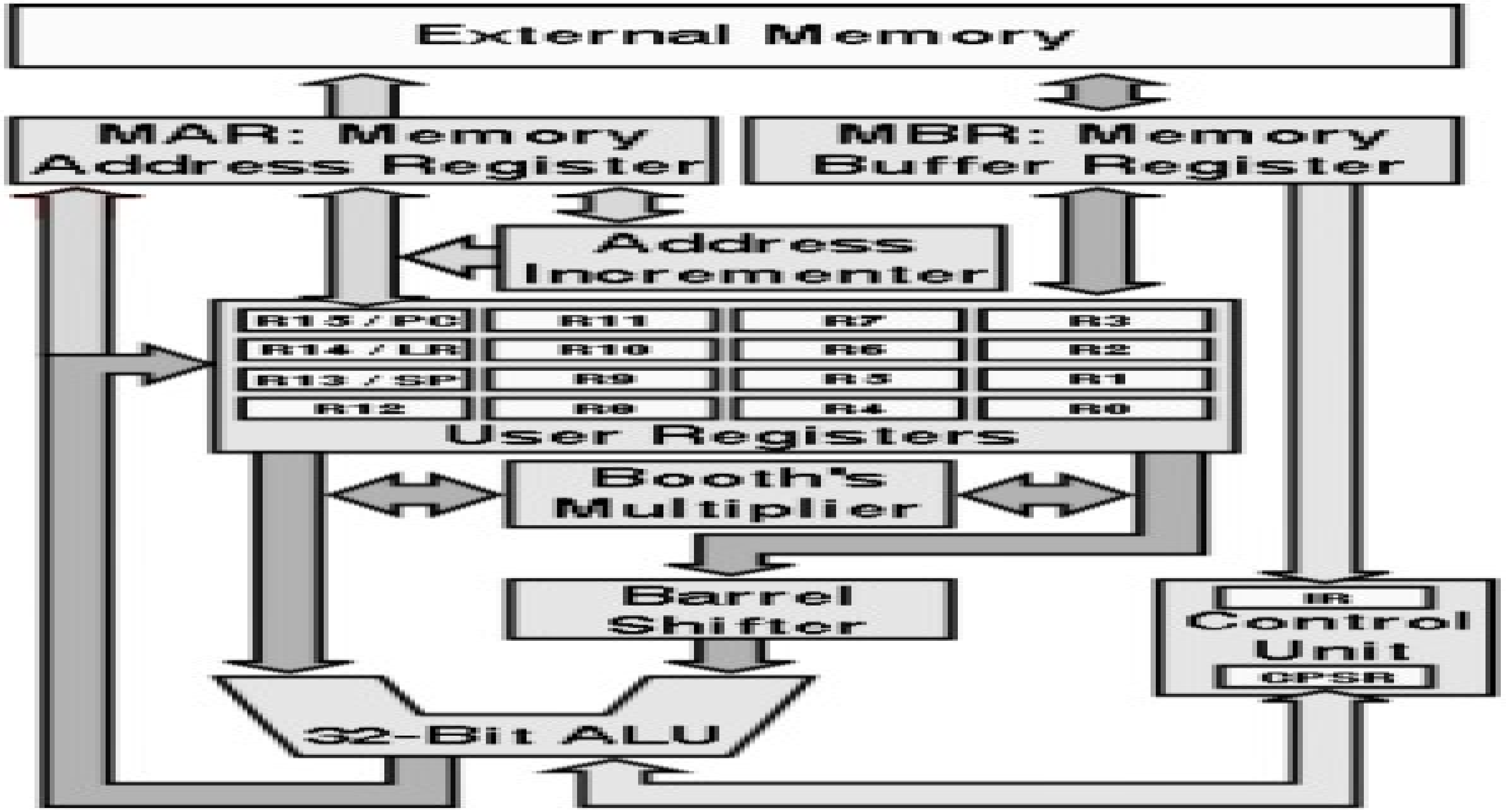
Registers


Privileged Modes
Exception Modes

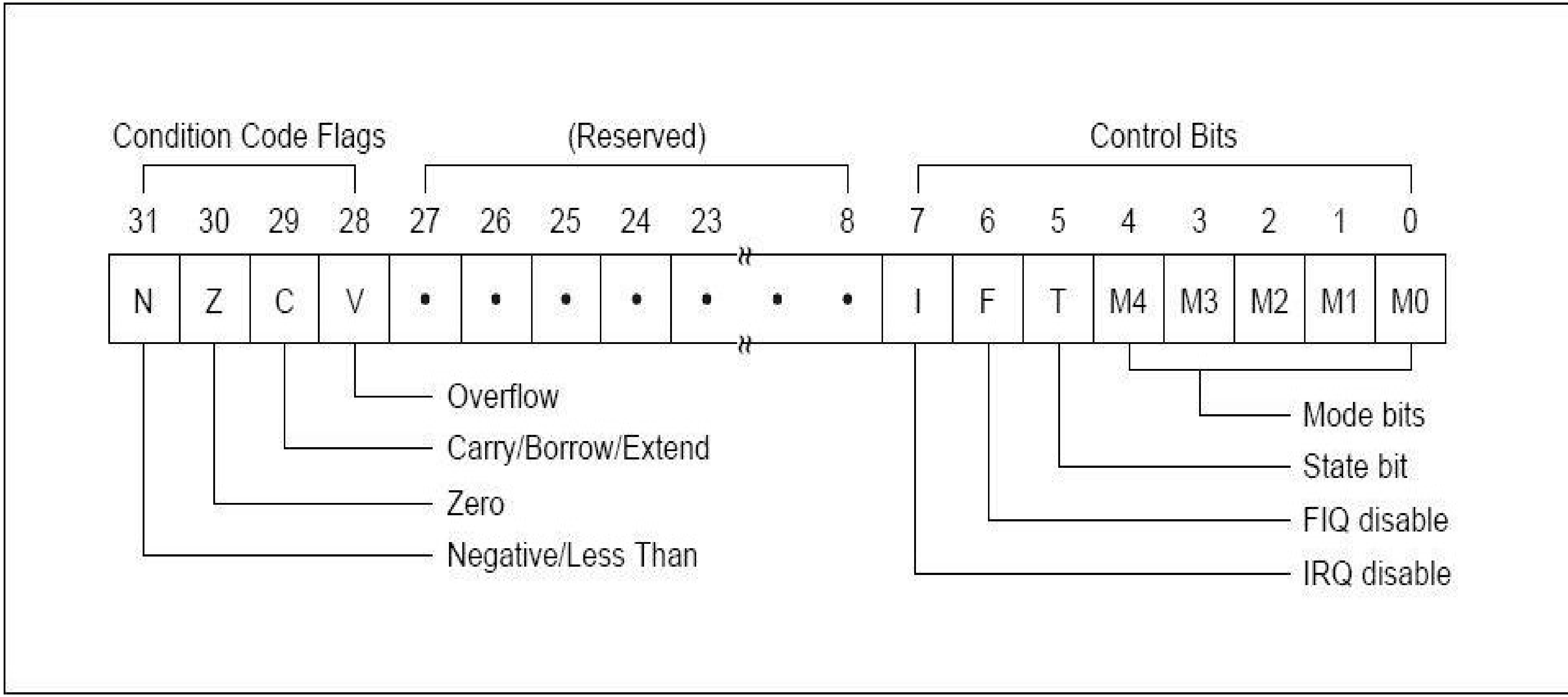
User	System	Supervisor	Abort	Undefined	Interrupt	Fast interrupt
r0	r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7	r7
r8	r8	r8	r8	r8	r8	r8_fiq
r9	r9	r9	r9	r9	r9	r9_fiq
r10	r10	r10	r10	r10	r10	r10_fiq
r11	r11	r11	r11	r11	r11	r11_fiq
r12	r12	r12	r12	r12	r12	r12_fiq
r13 sp	r13 sp	r13_svc	r13_abt	r13_und	r13_irq	r13_fiq
r14 lr	r14 lr	r14_svc	r14_abt	r14_und	r14_irq	r14_fiq
r15 pc	r15 pc	r15 pc	r15 pc	r15 pc	r15 pc	r15 pc
cpsr	cpsr	cpsr	cpsr	cpsr	cpsr	cpsr
–	–	spsr_svc	spsr_abt	spsr_und	spsr_irq	spsr_fiq

▨ Banked register

ARM Architecture



Current Program Status Register



Processor Modes

M[4:0]	Mode	Accessible registers
10000	User	PC, R14 to R0, CPSR
10001	FIQ	PC, R14_fiq to R8_fiq, R7 to R0, CPSR, SPSR_fiq
10010	IRQ	PC, R14_irq, R13_irq, R12 to R0, CPSR, SPSR_irq
10011	Supervisor	PC, R14_svc, R13_svc, R12 to R0, CPSR, SPSR_svc
10111	Abort	PC, R14_abt, R13_abt, R12 to R0, CPSR, SPSR_abt
11011	Undefined	PC, R14_und, R13_und, R12 to R0, CPSR, SPSR_und
11111	System	PC, R14 to R0, CPSR

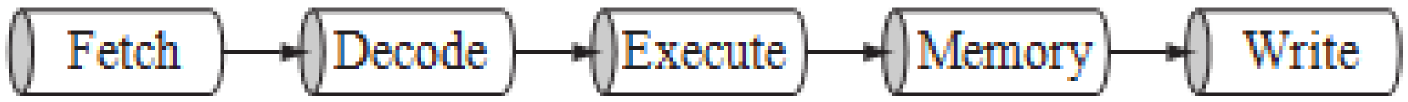
Instruction Pipeline

- The ARM uses a pipeline to increase the speed of the flow of instructions to the processor.
- This allows several operations to take place simultaneously.
- A three-stage pipeline is used, so instructions are executed in three stages:
- Fetch
- Decode
- Execute.

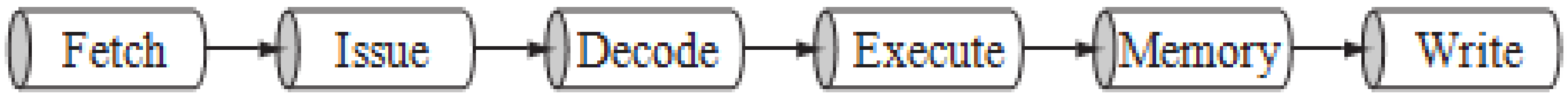
Instruction Pipeline

- During normal operation, while one instruction is being executed, its successor is being decoded, and a third instruction is being fetched from memory.
- The program counter points to the instruction being fetched rather than to the instruction being executed

Instruction Pipeline



ARM9 five-stage pipeline.



ARM10 six-stage pipeline.

Interrupts and Vector Table

- When an exception or interrupt occurs, the processor sets the pc to a specific memory address.
- The address is within a special address range called the vector table.
- The entries in the vector table are instructions that branch to specific routines designed to handle a particular exception or interrupt.
- The 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.

Interrupts and Vector Table

- **Reset vector** is the location of the first instruction 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** is called when you execute a SWI instruction. The SWI instruction is frequently used as the mechanism to invoke an operating system routine

Interrupts and Vector Table

- **Pre-fetch** 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 pre-fetch abort but is raised when ar 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.

Interrupts and Vector Table

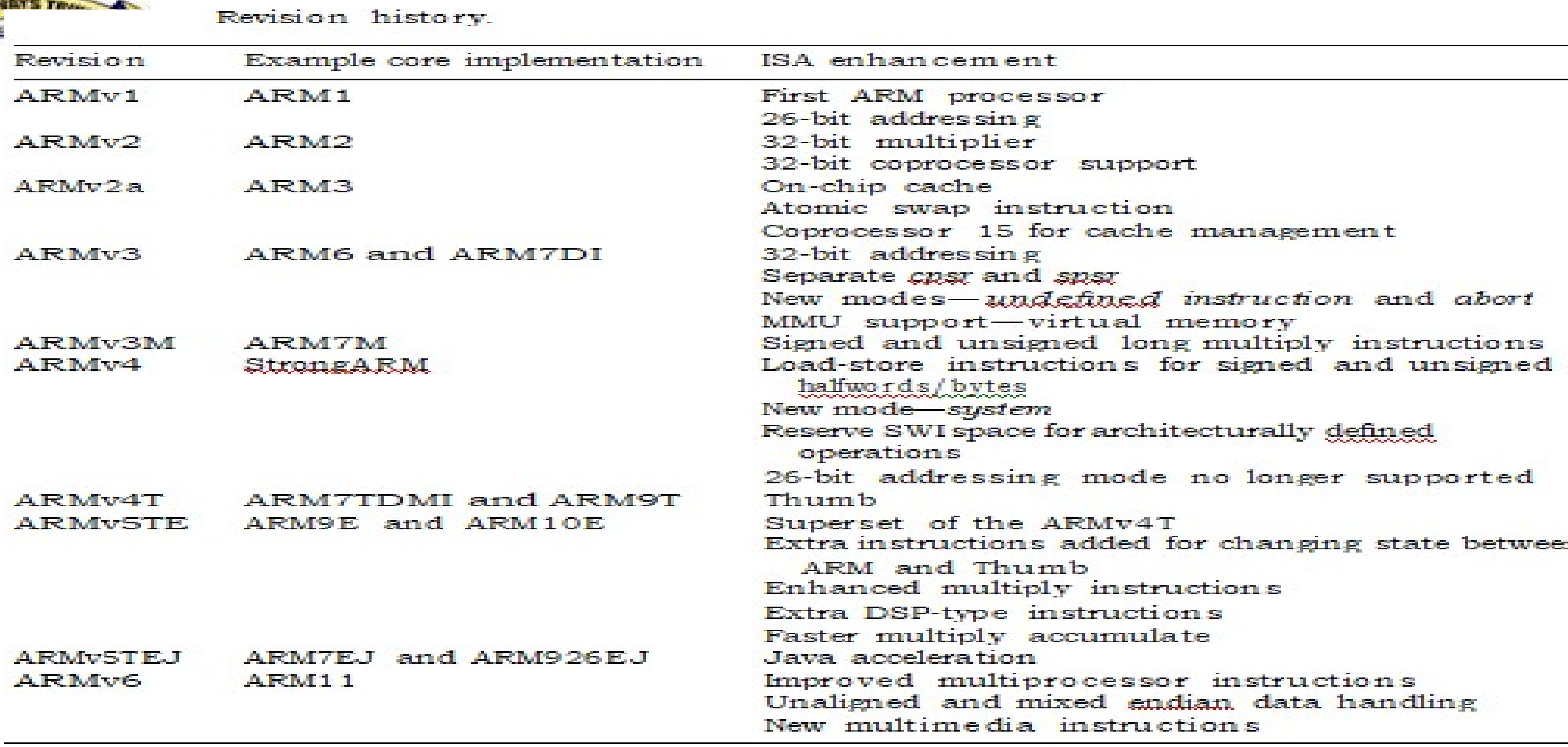
The vectortable.

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

ARM Processor Families

- ARM has designed a no. of processors grouped into different families according to the core used.
- ARM7
- ARM9
- ARM10
- ARM11
- Within each ARM family, no. of variations of memory management, cache etc.

ARM Processor Families


Revision history.

Revision	Example core implementation	ISA enhancement
ARMv1	ARM1	First ARM processor
		26-bit addressing
ARMv2	ARM2	32-bit multiplier
		32-bit coprocessor support
ARMv2a	ARM3	On-chip cache
		Atomic swap instruction
		Coprocessor 15 for cache management
ARMv3	ARM6 and ARM7DI	32-bit addressing
		Separate *cpsr* and *spsr*
		New modes—*undefined instruction* and *abort*
		MMU support—virtual memory
ARMv3M	ARM7M	Signed and unsigned long multiply instructions
ARMv4	StrongARM	Load-store instructions for signed and unsigned halfwords/bytes
		New mode—*system*
		Reserve SWI space for architecturally defined operations
		26-bit addressing mode no longer supported
ARMv4T	ARM7TDMI and ARM9T	Thumb
ARMv5TE	ARM9E and ARM10E	Superset of the ARMv4T
		Extra instructions added for changing state betwee ARM and Thumb
		Enhanced multiply instructions
		Extra DSP-type instructions
		Faster multiply accumulate
ARMv5TEJ	ARM7EJ and ARM926EJ	Java acceleration
ARMv6	ARM11	Improved multiprocessor instructions
		Unaligned and mixed endian data handling
		New multimedia instructions

ARM7 Family

- The ARM7 core has a Von Neumann–style architecture, where both data and instructions use the same bus.
- The core has a three-stage pipeline and executes the architecture ARMv4T instruction set.
- The ARM7TDMI was the first of a new range of processors introduced in 1995 by ARM.
- It is currently a very popular core and is used in many 32-bit embedded processors.
- The ARM7TDMI processor core has been licensed by many of the top semiconductor companies around the world and is the first core to include the Thumb instruction set.

ARM7 Family

- One significant variation in the ARM7 family is the ARM7TDMI-S.
- The ARM7TDMI-S has the same operating characteristics as a standard ARM7TDMI but is also synthesizable.
- ARM720T is the most flexible member of the ARM7 family because it includes an MMU.
- The presence of the MMU means the ARM720T is capable of handling the Linux and Microsoft embedded platform operating systems
- Another variation is the ARM7EJ-S processor, also synthesizable.
- ARM7EJ-S is quite different since it includes a five-stage pipeline.

ARM9 Family

- The ARM9 family was announced in 1997. Because of its five-stage pipeline, the ARM9 processor can run at higher clock frequencies than the ARM7 family.
- The extra stages improve the overall performance of the processor.
- The memory system has been redesigned to follow the Harvard architecture, which separates the data D and instruction I buses.
- The first processor in the ARM9 family was the ARM920T, which includes a separate D I cache and an MMU.
- ARM922T is a variation on the ARM920T but with half the D I cache size.

ARM9 Family

- The ARM940T includes a smaller D I cache and an MPU.
- The ARM940T is designed for applications that do not require a platform operating system.
- The next processors in the ARM9 family were based on the ARM9E-S core. This core is a synthesizable version of the ARM9 core.
- There are two variations: the ARM946E-S and the ARM966E-S.
- The latest core in the ARM9 product line is the ARM926EJ-S synthesizable processor core, announced in 2000.

ARM10 Family

- The ARM10, announced in 1999, was designed for performance.
- It extends the ARM9 pipeline to six stages.
- It also supports an optional vector floating-point (VFP) unit, which adds a seventh stage to the ARM10 pipeline.
- The VFP significantly increases floating-point performance and is compliant with the IEEE 754.1985 floating-point standard.
- The ARM1020E has separate 32K D I caches, optional vector floating-point unit, and an MMU.
- The ARM1020E also has a dual 64-bit bus interface for increased performance.

ARM11 Family

- The ARM1136J-S, announced in 2003, was designed for high performance and power- efficient applications.
- ARM1136J-S was the first processor implementation to execute architecture ARMv6 instructions.
- It incorporates an eight-stage pipeline with separate load- store and arithmetic pipelines.
- Included in the ARMv6 instructions are single instruction multiple data (SIMD) extensions for media processing, specifically designed to increase video processing performance.

Instruction Set

- All ARM instructions are 32 bits wide (except the compressed 16-bit Thumb Instructions) and are aligned on 4-byte boundaries in memory.
- The most notable features of the ARM instruction set are:
- The load-store architecture;
- 3-address data processing instructions (that is, the two source operand registers and the result register are all independently specified);
- conditional execution of every instruction

Instruction Set

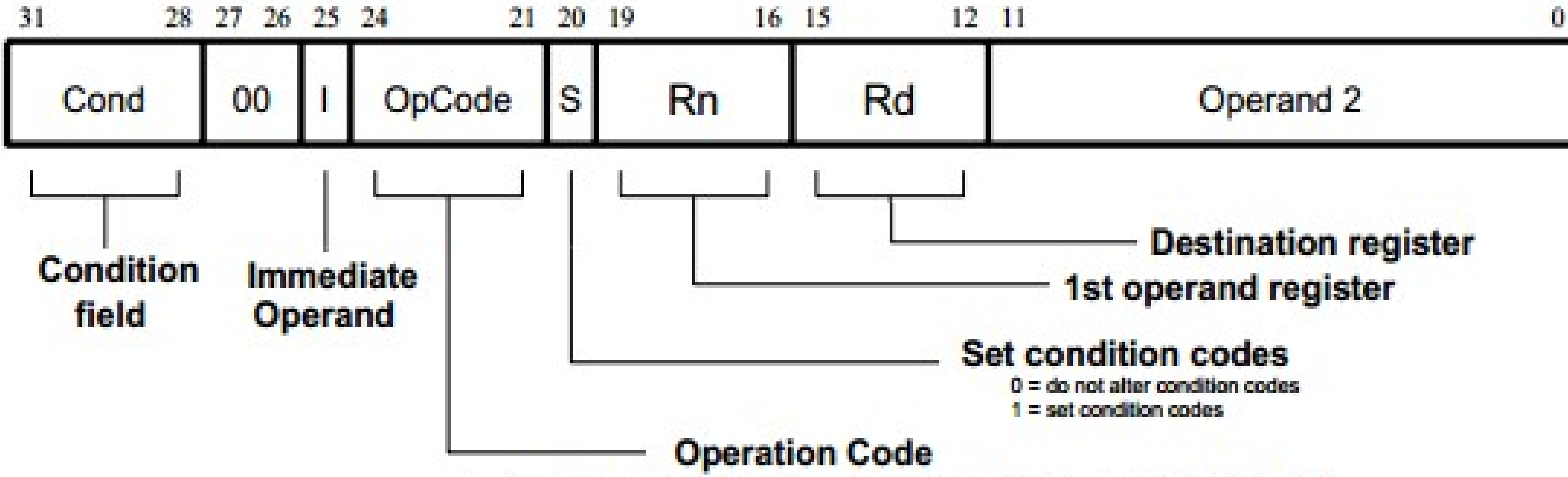
✓ the inclusion of very powerful load and store multiple register instructions;

✓ the ability to perform a general shift operation and a general ALU operation in a single instruction that executes in a single clock cycle;

✓ open instruction set extension through the coprocessor instruction set, including adding new registers and data types to the programmer's model;

✓ a very dense 16-bit compressed representation of the instruction set in the Thumb architecture.

Instruction Format



0000 = AND - Rd:= Op1 AND Op2	1000 = TST - set condition codes on Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2	1001 = TEQ - set condition codes on Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2	1010 = CMP - set condition codes on Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1	1011 = CMN - set condition codes on Op1 + Op2
0100 = ADD - Rd:= Op1 + Op2	1100 = ORR - Rd:= Op1 OR Op2
0101 = ADC - Rd:= Op1 + Op2 + C	1101 = MOV - Rd:= Op2
0110 = SBC - Rd:= Op1 - Op2 + C - 1	1110 = BIC - Rd:= Op1 AND NOT Op2
0111 = RSC - Rd:= Op2 - Op1 + C - 1	1111 = MVN - Rd:= NOT Op2

ARM assembly language

- Fairly standard assembly language:

```
        LDR   r0,[r8]     ; a  comment
label   ADD   r4,r0,r1
```

ARM data types

- **32-bit word.**
- **Word can be divided into four 8-bit bytes.**
- **ARM addresses can be 32 bits long.**
- **Address refers to byte.**
  - Address 4 starts at byte 4.
- **Can be configured at power-up as either little- or bit-endian mode.**

Instruction Set

- The ARM processor is very easy to program at the assembly level
- In this part, we will
  - **Look at ARM instruction set and assembly language programming at the user level**

Notable Features of ARM Instruction Set

- The load-store architecture
- 3-address data processing instructions
- **Conditional execution of every instruction**
- The inclusion of every powerful load and store multiple register instructions
- Single-cycle execution of all instruction
- Open coprocessor instruction set extension

Conditional Execution (1)

- One of the ARM's most interesting features is that each instruction is **conditionally executed**
- In order to indicate the ARM's conditional mode to the assembler, all you have to do is to append the appropriate condition to a mnemonic

```
     CMP          r0, #5
     BEQ          BYPASS
         ADD         r1, r1, r0
         SUB         r1, r1, r2
BYPASS
        …
```

$\Longrightarrow$

```
     CMP          r0, #5
     ADDNE        r1, r1,   r0
     SUBNE        r1, r1,   r2
…
```

Conditional Execution (2)

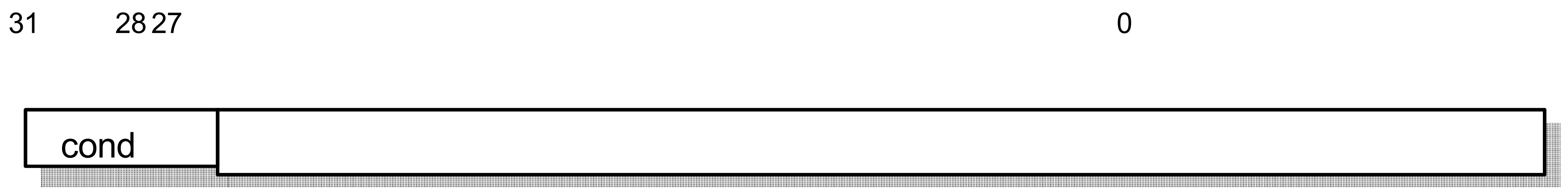
- The conditional execution code is faster and smaller

```
; if ((a==b) && (c==d))              e++;
;
; a is in register r0
; b is in register r1
; c is in register r2
; d is in register r3
; e is in register r4

     CMP       r0, r1  r2, r3
     CMPEQ     r4, r4, #1
     ADDEQ
```

The ARM Condition Code Field

- Every instruction is conditionally executed
- Each of the 16 values of the condition field causes the instruction to be executed or skipped according to **the values of the N, Z, C and V flags in the CPSR**



N: Negative **Z: Zero** **C: Carry** **V: oVerflow**

ARM Condition Codes

pcode [3 :28]	Mnemonic	Interpretation extensio	Status flag state for execution
00	EQ	Equal / equals zero	Z set
01	NE	Not equal	Z clear
10	CS/HS	Carry set / unsigned higher or same	C set
11	CC/LO	Carry clear / unsigned lower	C clear
00	MI	Minus / negative	N set
01	PL	Plus / positive or zero	N clear
10	VS	Overflow	V set
11	VC	No overflow	V clear
00	HI	Unsigned higher	C set and Z clear
01	LS	Unsigned lower or same	C clear or Z set
10	GE	Signed greater than or equal	N equals V
11	LT	Signed less than	N is not equal to V
00	GT	Signed greater than	Z clear and N equals V
01	LE	Signed less than or equal	Z set or N is not equal to V
10	AL	Always	any
11	NV	Never (do not use!)	none

Condition Field

- In ARM state, all instructions are conditionally executed according to the CPSR condition codes and the instruction’s condition field
- Fifteen different conditions may be used
- **“Always” condition**
  - Default condition
  - May be omitted
- **“Never” condition**
  - The sixteen (1111) is reserved, and must not be used
  - May use this area for other purposes in the future

ARM Instruction Set

- Data processing instructions
- Data transfer instructions
- Control flow instructions
- Writing simple assembly language programs

ARM Instruction Set

- **Data processing instructions**
- Data transfer instructions
- Control flow instructions
- Writing simple assembly language programs

Data processing instructions

- Enable the programmer to perform **arithmetic** and **logical** operations on data values in registers
- The applied rules
  - All operands are 32 bits wide and come from registers or are specified as literals in the instruction itself
  - The result, if there is one, is 32 bits wide and is placed in a register

    (An exception: long multiply instructions produce a 64 bits result)
  - Each of the operand registers and the result register are independently specified in the instruction

    (This is, the ARM uses a ‘**3-address**’ format for these instruction)

Simple Register Operands

```
ADD  r0, r1, r2                ; r0 := r1 + r2
```

The semicolon here indicates that everything to the right of it is a comment and should be ignored by the assembler

The values in the register may be considered to be unsigned integer or signed 2's-complement values

Arithmetic Operations

- These instructions perform binary arithmetic on two 32- bit operands
- The carry-in, when used, is the current value of the C bit in the CPSR

ADD r0, r1, r2	r0 := r1 + r2
ADC r0, r1, r2	r0 := r1 + r2 + C
SUB r0, r1, r2	r0 := r1 – r2
SBC r0, r1, r2	r0 := r1 – r2 + C – 1
RSB r0, r1, r2	r0 := r2 – r1
RSC r0, r1, r2	r0 := r2 – r1 + C – 1

Bit-Wise Logical Operations

- These instructions perform the specified boolean logic operation on each bit pair of the input operands

```
r0[i] := r1[i] OP_logic r2[i]     for i in [0..31]
```

AND r0, r1, r2	r0 := r1 AND r2
ORR r0, r1, r2	r0 := r1 OR r2
EOR r0, r1, r2	r0 := r1 XOR r2
BIC r0, r1, r2	r0 := r1 AND (NOT r2)

- **BIC stands for ‘bit clear’**
- **Every ‘1’ in the second operand clears the corresponding bit in the first operand**

Example: BIC Instruction

- r1 = 0x11111111
- r2 = 0x01100101
- **BIC r0, r1, r2**
- r0 = 0x10011010

Register Movement Operations

- These instructions ignore the first operand, which is omitted from the assembly language format, and simply move the second operand to the destination

MOV r0, r2	r0 := r2
MVN r0, r2	r0 := NOT r2

The ‘**MVN**’ mnemonic stands for ‘**move negated**’

Comparison Operations

- These instructions do not produce a result, but just set the condition code bits (N, Z, C, and V) in the CPSR according to the selected operation

CMP r1, r2	**compare**	set cc on r1 – r2
CMN r1, r2	**compare negated**	set cc on r1 + r2
TST r1, r2	**bit test**	set cc on r1 AND r2
TEQ r1, r2	**test equal**	set cc on r1 XOR r2

Immediate Operands

- If we wish to add a constant to a register, we can replace the second source operand with an immediate value

```
ADD    r3, r3, #1                ; r3 := r3 + 1
AND    r8, r7, #&ff              ; r8 := r7[7:0]
```

A constant preceded by '#'

A hexadecimal by putting '&' after the '#'

Shifted Register Operands (1)

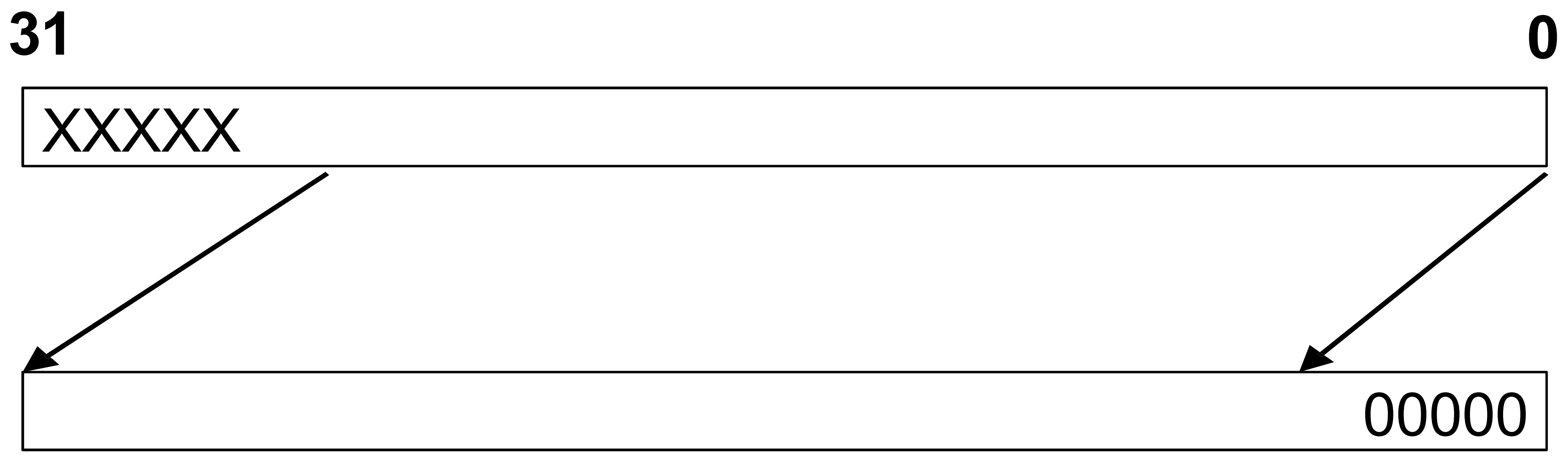
- These instructions allows **the second register operand to be subject to a shift operation** before it is combined with the first operand

```
ADD    r3, r2, r1, LSL #3     ; r3 := r2 + 8 * r1
```

- They are still single ARM instructions, executed in a single clock cycle
- Most processors offer shift operations as separate instructions, but the ARM combines them with a general ALU operation in a single instruction

Shifted Register Operands (2)

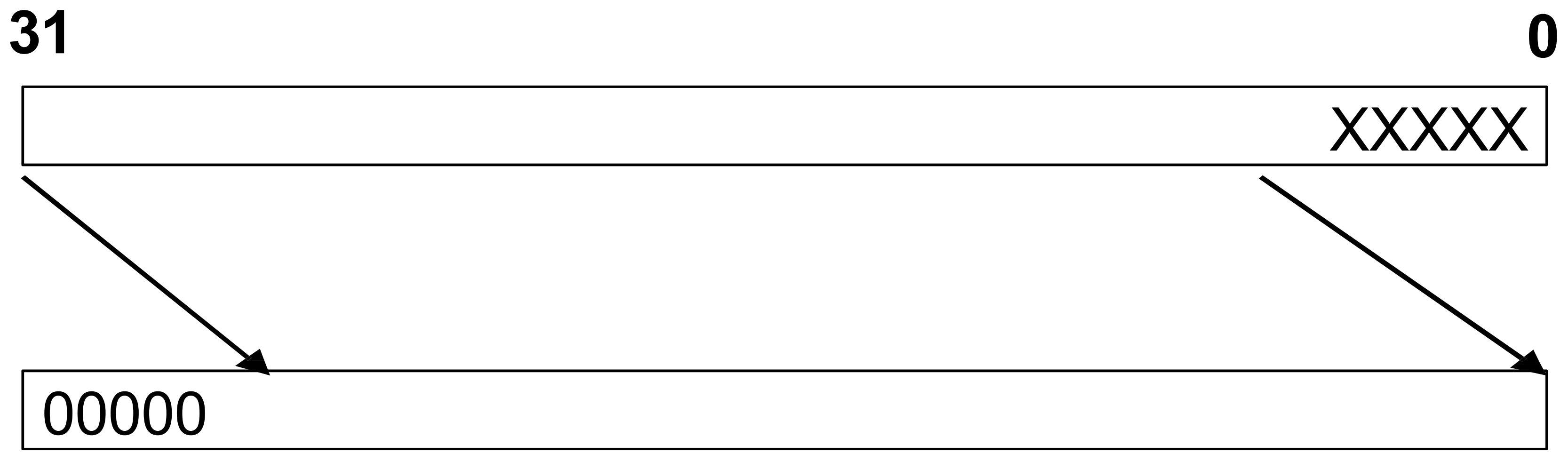
LSL	logical shift left by 0 to 31	Fill the vacated bits at the LSB of the word with zeros
ASL	arithmetic shift left	A synonym for LSL



LSL #5

Shifted Register Operands (3)

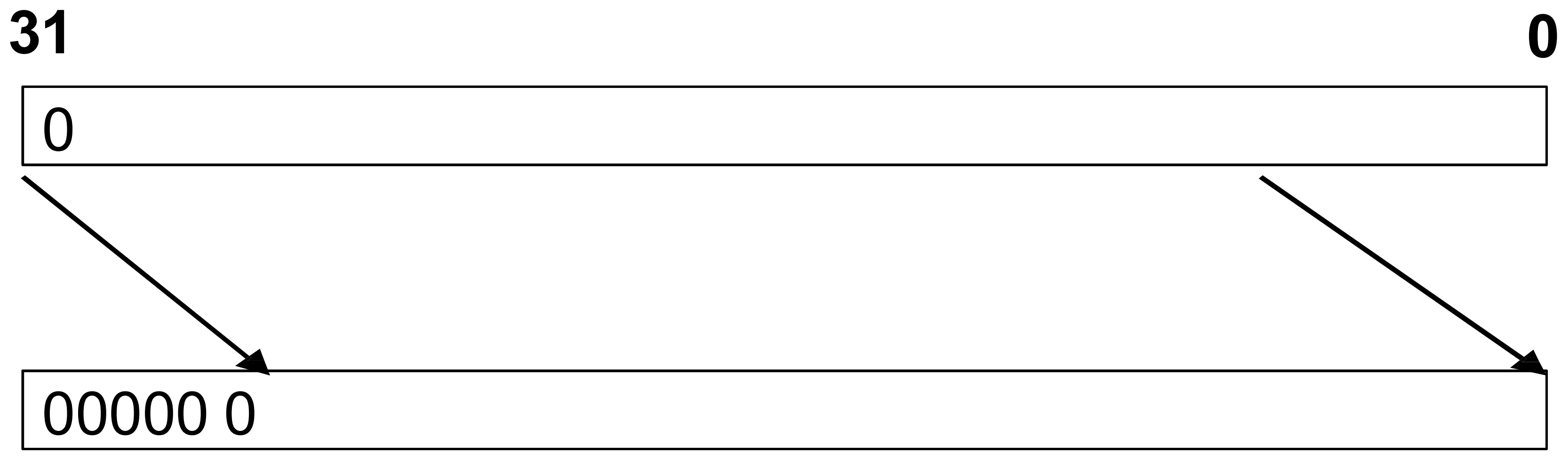
LSR	logical shift right by 0 to 31	Fill the vacated bits at the MSB of the word with zeros



LSR #5

Shifted Register Operands (4)

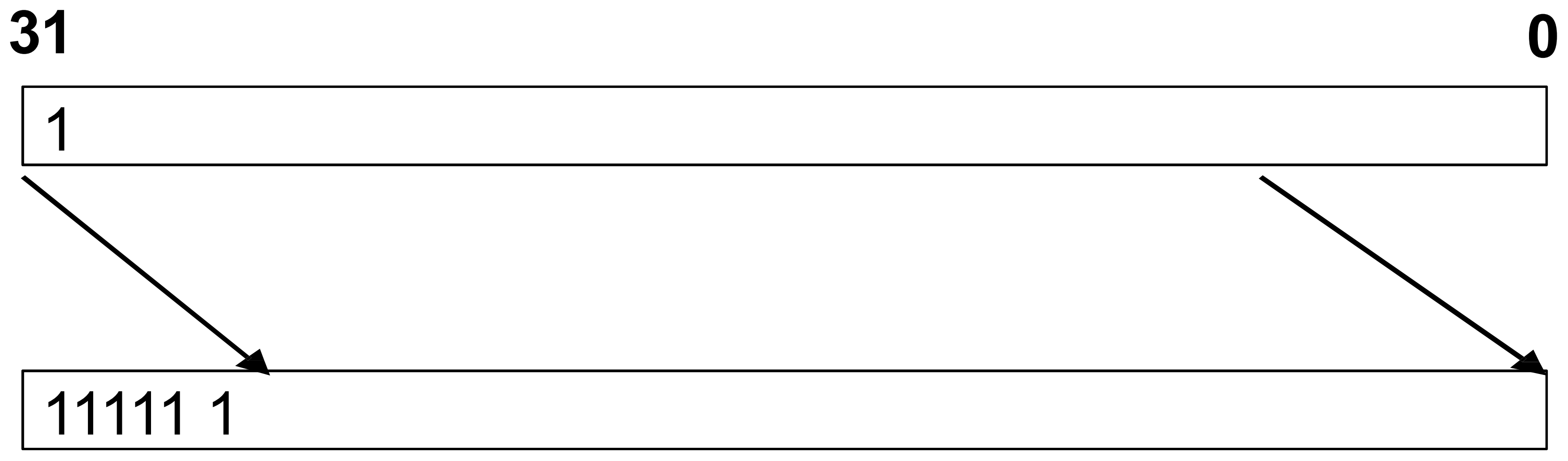
ASR	arithmetic shift right by 0 to 31	Fill the vacated bits at the MSB of the word with zero (source operand is positive)



ASR #5 ;positive operand

Shifted Register Operands (5)

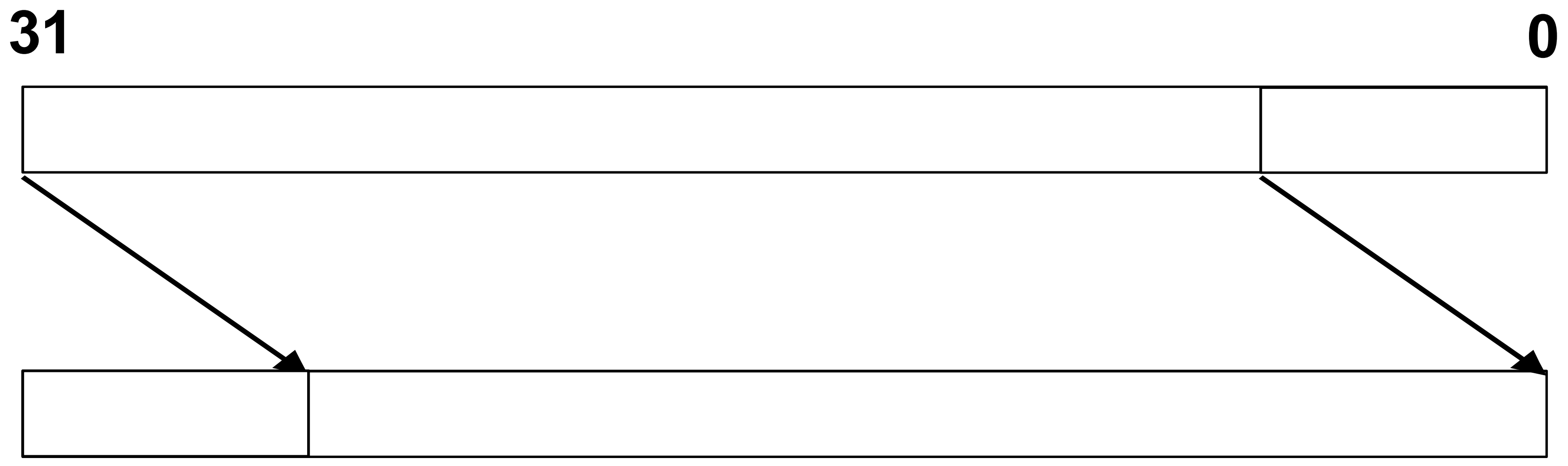
ASR	arithmetic shift right by 0 to 31	Fill the vacated bits at the MSB of the word with one (source operand is negative)



ASR #5 ;negative operand

Shifted Register Operands (6)

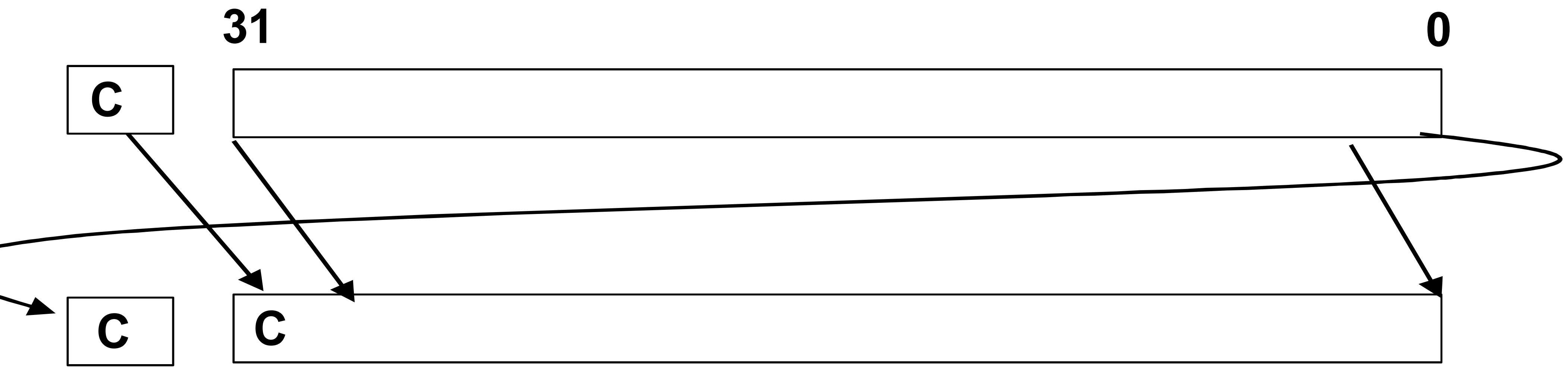
ROR	Rotate right by 0 to 31	The bits which fall off the LSB of the word are used to fill the vacated bits at the MSB of the word



ROR #5

Shifted Register Operands (7)

RRX	Rotate right extended by 1 place	The vacated bit (bit 31) is filled with the old value of the C flag and the operand is shifted one place to the right



RRX

Shifted Register Operands (8)

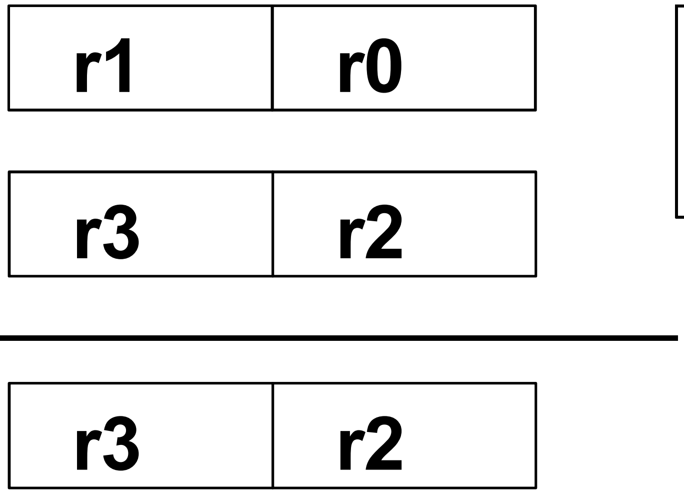
- It is possible to use a register value to specify the number of bits the second operand should be shifted by
- Ex:

```
ADD    r5, r5, r3, LSL r2    ; r5:=r5+r3*2^r2
```

- Only the bottom 8 bits of r2 are significant

Setting the Condition Codes

- Any data processing instruction can set the condition codes (N, Z, C, and V) if the programmer wishes it to
- Ex: 64-bit addition



```
ADDS          r2, r2, r0 ; 32-bit carry out->C
ADC           r3, r3, r1 ; C is added into
                         ; high word
```

Adding ‘S’ to the opcode, standing for ‘Set condition codes’

Multiplies (1)

- A special form of the data processing instruction supports multiplication
- Some important differences
  - Immediate second operands are not supported
  - The result register must not be the same as the first source register
  - If the ‘S’ bit is set, the C flag is meaningless

```
MUL    r4, r3, r2    ; r4 := (r3 x r2)[31:0]
```

Multiplies (2)

- The multiply-accumulate instruction

```
MLA    r4, r3, r2, r1    ; r4 := (r3 x r2 + r1)[31:0]
```

- In some cases, it is usually more efficient to use a short series of data processing instructions
- Ex: multiply r0 by 35

```
  ; move 35 to r1
 MUL    r3, r0, r1 ; r3 := r0 x 35
```

OR

```
 ADD    r0, r0, r0, LSL #2 ; r0’  := 5 x r0
 RSB    r0, r0, r0, LSL #3 ; r0’’:= 7 x r0’
```

ARM Instruction Set

- Data processing instructions
- **Data transfer instructions**
- Control flow instructions
- Writing simple assembly language programs

Addressing mode

- The ARM data transfer instructions are all based around **register-indirect addressing**
  - Based-plus-offset addressing
  - Based-plus-index addressing

```
LDR        r0, [r1]            ; r0 := mem32[r1]
STR        r0, [r1]            ; mem32[r1] := r0
```

Register-indirect addressing

Data Transfer Instructions

- Move data between ARM registers and memory
- Three basic forms of data transfer instruction
  - Single register load and store instructions
  - Multiple register load and store instructions
  - Single register swap instructions

Single Register Load / Store Instructions (1)

- These instructions provide the most flexible way to transfer single data items between an ARM register and memory
- The data item may be a **byte**, a **32-bit word**, **16- bit half-word**

```
LDR         r0, [r1]              ; r0 := mem32[r1]
STR         r0, [r1]              ; mem32[r1] := r0
```

Register-indirect addressing

Single Register Load / Store Instructions (2)

DR	Load a word into register	Rd ←mem32[address]
TR	Store a word in register into memory	Mem32[address] ←Rd
DRB	Load a byte into register	Rd ←mem8[address]
TRB	Store a byte in register into memory	Mem8[address] ←Rd
DRH	Load a half-word into register	Rd ←mem16[address]
TRH	Store a half-word in register into memory	Mem16[address] ←Rd
DRSB	Load a signed byte into register	Rd ←signExtend(mem8[address])
DRSH	Load a signed half-word into register	Rd ←signExtend(mem16[address])

Base-plus-offset Addressing (1)

- **Pre-indexed addressing mode**
  - It allows one base register to be used to access a number of memory locations which are in the same area of memory

```
LDR    r0, [r1, #4]    ; r0 := mem32[r1 + 4]
```

Base-plus-offset Addressing (2)

- **Auto-indexing (Preindex with writeback)**
  - No extra time
  - The time and code space cost of the extra instruction are avoided

```
LDR      r0,    [r1,    #4]!          ; r0  := mem32[r1    + 4]
                                      ; r1  := r1 + 4
```

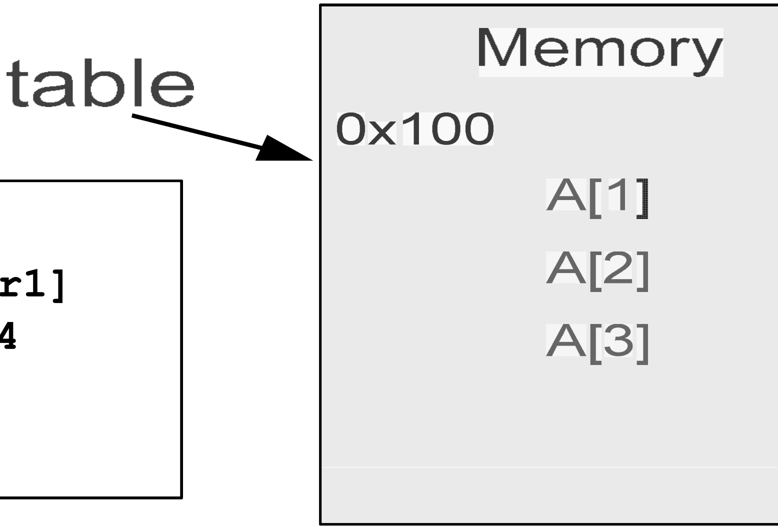
The exclamation “!” mark indicates that the instruction should update the base register after initiating the data transfer

Base-plus-offset Addressing (3)

- **Post-indexed addressing mode**
  - The exclamation “!” is not needed

```
LDR     r0,    [r1],    #4          ; r0  := mem32[r1]
                                    ; r1  := r1 + 4
```

Application



```
          ADR      r1, table
LOOP      LDR      r0, [r1]                 ; r0  :=  mem32[r1]
          ADD      r1, r1, #4               ; r1  :=  r1 + 4
          ;do some operation on r0
          …
```

```
          ADR   r1, table
LOOP      LDR         r0, [r1], #4      ; r0   :=  mem32[r1]
                                        ; r1   :=  r1 + 4
          ;do        some operation on  r0
          …
```

Multiple Register Load / Store Instructions (1)

- Enable large quantities of data to be transferred more efficiently
- They are used for procedure entry and exit to save and restore workspace registers
- Copy blocks of data around memory

```
LDMIA      r1, {r0, r2, r5}           ; r0  := mem32[r1]
                                      ; r2  := mem32[r1 + 4]
                                      ; r5  := mem32[r1 + 8]
```

The base register r1 should be word-aligned

Multiple Register Load / Store Instructions (2)

LDM	Load multiple registers
STM	Store multiple registers

Addressing mode	**Description**	**Starting 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*Rn+4	Rn	Rn-4*N
DB	Decrement Before	Rn-4*N	Rn-4	Rn-4*N

Addressing mode for multiple register load and store instructions

Example (1)

r0 → 0x100

address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x200	50
0x204	60

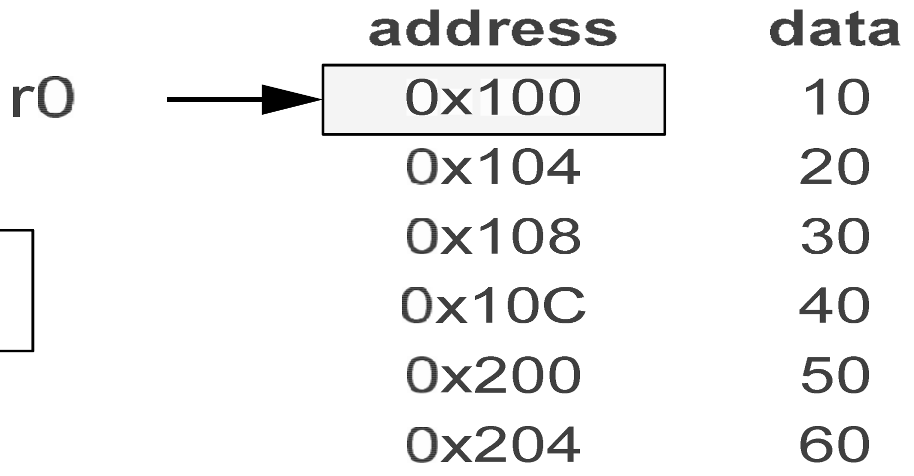
```
LDMIA          r0, {r1, r2,      r3}
OR
LDMIA          r0, {r1-r3}
```

⇩

```
 r1   :=  10
 r2   :=  20
 r3   :=  30

 r0   :=  0x100
```

Example (2)



address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x200	50
0x204	60

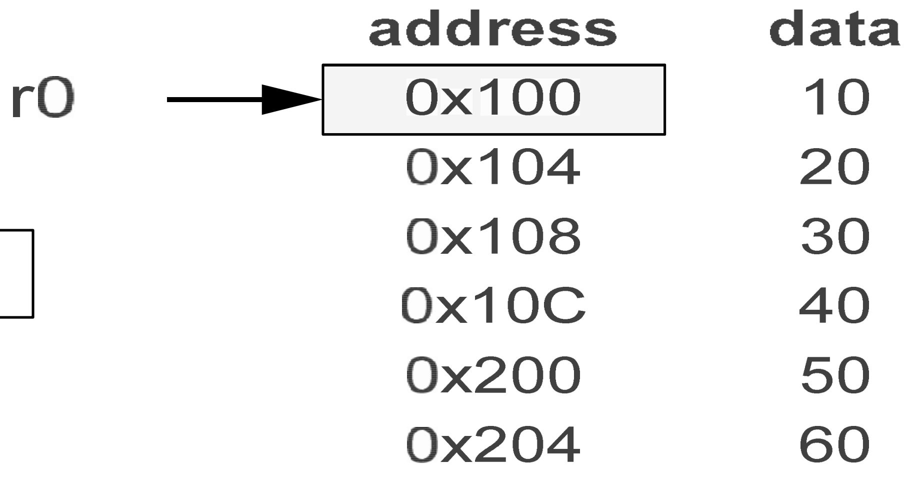
```
LDMIA      r0!, {r1, r2, r3}
```

```
r1   :=  10
r2   :=  20
r3   :=  30

r0   :=  0x10C
```

Example (3)

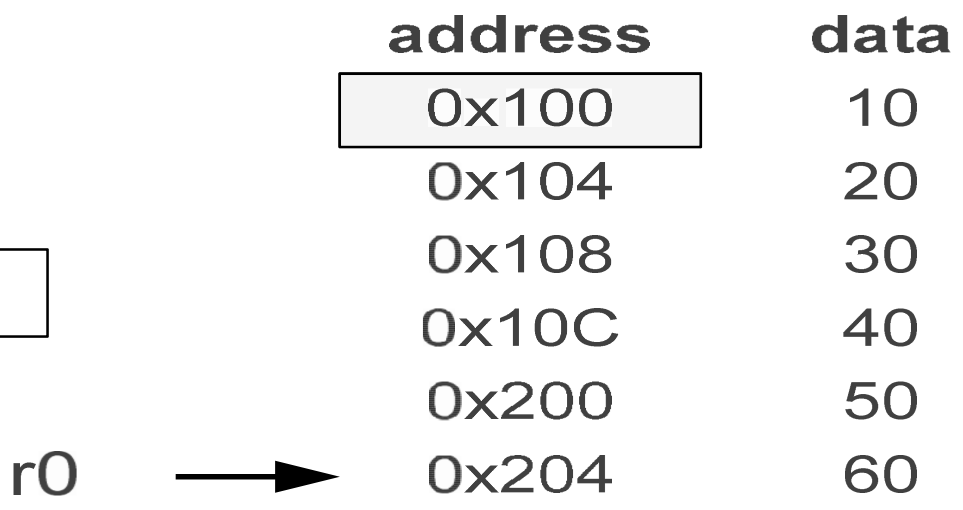
```
LDMIB    r0!, {r1, r2, r3}
```



```
r1   :=  20
r2   :=  30
r3   :=  40

r0   :=  0x10C
```

Example (4)



address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x200	50
0x204	60

```
LDMDA    r0!, {r1, r2, r3}
```

```
r1   :=  40
r2   :=  50
r3   :=  60

r0   :=  0x108
```

Example (5)

```
LDMDB   r0!, {r1, r2, r3}
```

```
r1   :=  30
r2   :=  40
r3   :=  50

r0   :=  0x108
```

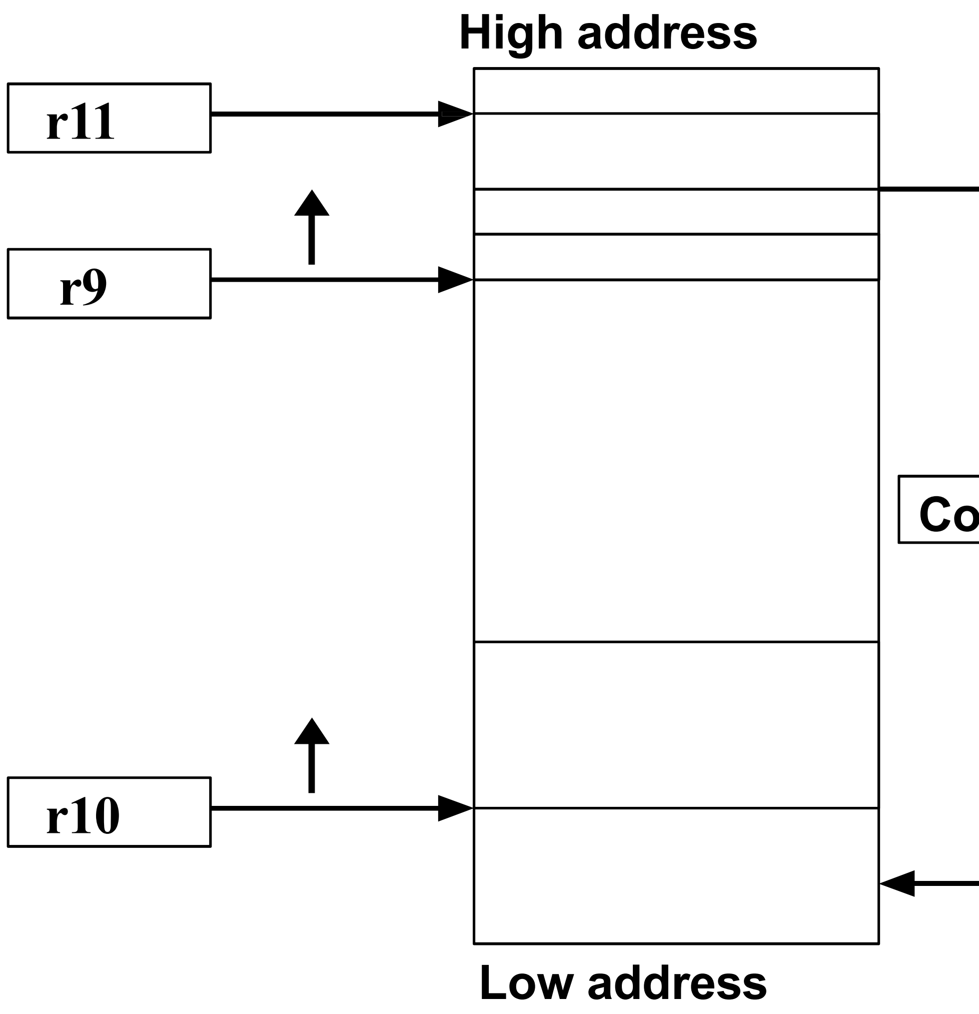
	address	data
	0x100	10
	0x104	20
	0x108	30
	0x10C	40
	0x200	50
r0 →	0x204	60

Application

Copy a block of memory

```
r9    begin address of source data
r10   begin address of target
r11   end address of source data

P
     LDMIA    r9! , {r0-r7}
     STMIA    r10!, {r0-r7} CMP      r9
              , r11
     BNE      LOOP
```



Application: Stack Operations

- ARM use multiple load-store instructions to operate stack
  - **POP**: multiple load instructions
  - **PUSH**: multiple store instructions

The Stack (1)

- Stack grows up or grows down
  - Ascending, ‘A’
  - Descending, ‘D’
- Full stack, ‘F’: sp points to the last used address in the stack
- Empty stack, ‘E’: sp points to the first unused address in the stack

The Stack (2)

The mapping between the stack and block copy views of the multiple load and store instructions

Addressing mode	POP	=LDM	PUSH	=STM
FA	LDMFA	LFMFA	STMFA	STMIB
FD	LDMFD	LDMIA	STMFD	STMDB
EA	LDMEA	LDMDB	STMEA	STMIA
ED	LDMED	LDMIB	STMED	STMDA

Single Register Swap Instructions (1)

- Allow a value in a register to be exchanged with a value in memory
- Effectively do both a load and a store operation in one instruction
- They are little used in user-level programs
- Atomic operation
- Application
  - Implement semaphores (multi-threaded / multi-processor environment)

Single Register Swap Instructions (2)

```
SWP{B}  Rd, Rm, [Rn]
```

SWP	WORD exchange	tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp
SWPB	Byte exchange	tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp

Example

r0: 123456

r1: 111111

r2: 0x108

address	data
0x100	10
0x104	20
0x108	30

```
SWP  r0, r1, [r2]
```

r0: 30

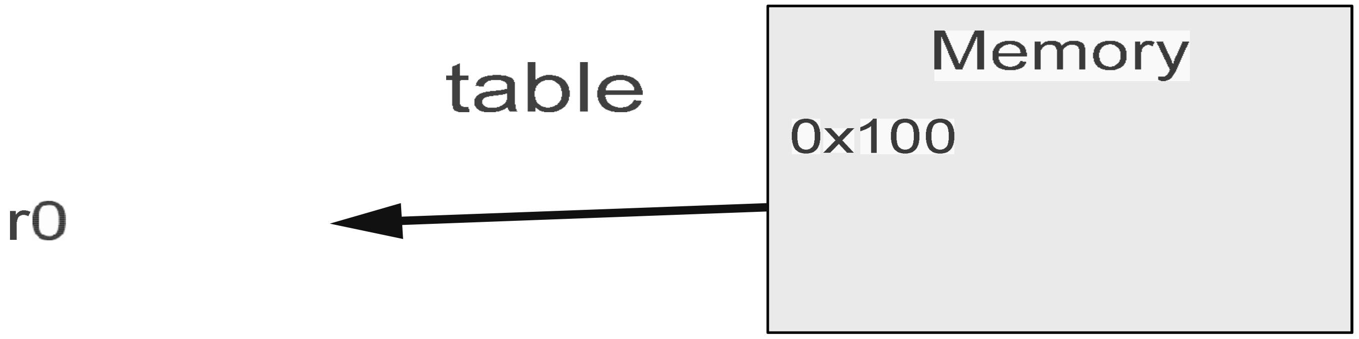
r1: 111111

r2: 0x108

address	data
0x100	10
0x104	20
0x108	111111

Load an Address into Register (1)

- The **ADR** (load address into register) instruction to load a register with a 32-bit address
- Example
  - **ADR** r0,table
  - Load the contents of register r0 with the 32-bit address "table"



Load an Address into Register (2)

- ADR is a **pseudo** instruction
- Assembler will transfer pseudo instruction into a sequence of appropriate normal instructions
- Assembler will transfer ADR into a single ADD, or SUB instruction to load the address into a register.

ARM Instruction Set

- Data processing instructions
- Data transfer instructions
- **Control flow instructions**
- Writing simple assembly language programs

Control Flow Instructions

- Determine which instructions get executed next

```
        B       LABEL
        …
        …
LABEL   …
```

```
        MOV     r0, #0          ;  initialize counter
LOOP    …
        ADD     r0, r0,    #1   ;  increment loop counter
        CMP     r0, #10         ;  compare with limit
        BNE     LOOP            ;  repeat if not equal
        …                       ;  else fall through
```

Branch Conditions

Branch	Interpretation	Normal uses
B BAL	Unconditional Always	Always take this branch Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Branch Instructions

B		PC=label

BL		PC=label
		LR=BL

BX		PC=Rm & 0xfffffffe, T=Rm & 1

BLX		PC=label, T=1
		PC=Rm & 0xfffffffe, T=Rm & 1
		LR = BLX

Branch and Link Instructions (1)

- BL instruction save the return address into r14 (lr)

```
          BL      subroutine           ; branch to subroutine
          CMP     r1, #5               ; return to here
          MOVEQ   r1, #0
          …

subroutine                              ; subroutine entry point
        …
        MOV         pc, lr               ; return
```

Branch and Link Instructions (2)

- **Problem**
  - If a subroutine wants to call another subroutine, the original return address, **r14**, will be overwritten by the second BL instruction
- **Solution**
  - Push **r14** into a stack
  - The subroutine will often also require some work registers, the old values in these registers can be saved at the same time using a store multiple instruction

Branch and Link Instructions (3)

```
     BL       SUB1     ; branch to subroutine SUB1
     …
```

```
SUB1
     STMFD        r13!,       {r0-r2,r14}     ;  save work & link register
     BL           SUB2
     …
     LDMFD        r13!,       {r0-r2, pc}     ;  restore work register and
                                              ;  return
```

```
SUB2
    …
    MOV         pc, r14      ; copy r14 into r15 to return
```

Jump Tables (1)

- A programmer sometimes wants to call one of a set of subroutines, the choice depending on a value computed by the program

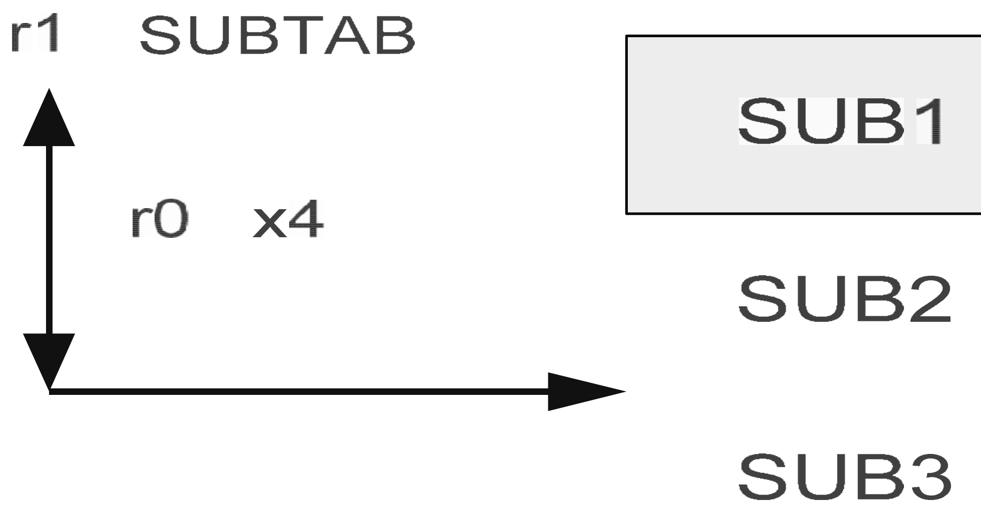
Note: slow when the list is long, and all subroutines are equally frequent

```
    BL          JUMPTAB
    ..
JUMPTAB
           CMP  r0, #0
           BEQ  SUB0
           CMP  r0, #1
           BEQ  SUB1
           CMP  r0, #2
    BEQ         SUB2
    ..
```

Jump Tables (2)

- "**DCD**" directive instructs the assembler to reserve a word of store and to initialize it to the value of the expression to the right

```
    BL          JUMPTAB
    ..
UMPTAB
    ADR    r1, SUBTAB CMP
           r0, #SUBMAX
     LDRLS pc, [r1, r0, LSL #2]
    B           ERROR
UBTAB
    DCD    SUB0
    DCD    SUB1
    DCD    SUB2
    ..
```



Supervisor Calls

- SWI: SoftWare Interrupt
- The supervisor calls are implemented in system software
  - They are probably different from one ARM system to another
  - Most ARM systems implement a common subset of calls in addition to any specific calls required by the particular application

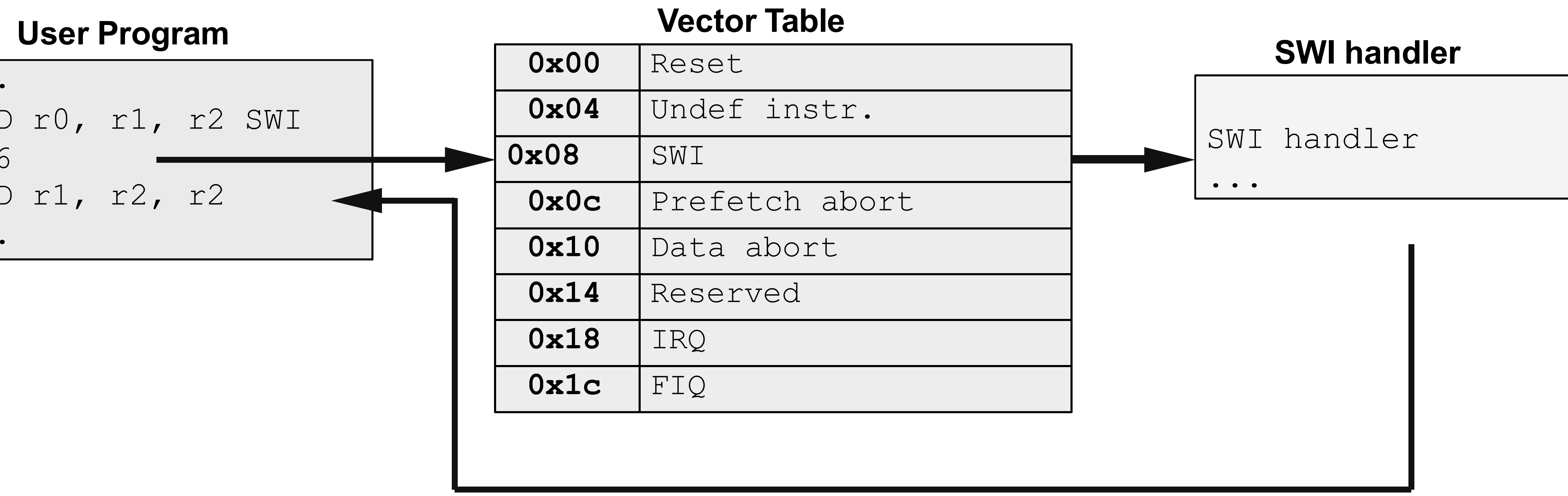
```
; This routine sends the character in the bottom
; byte of r0 to the use display device
```

```
    SWI     SWI_WriteC   ; output r0[7:0]
```

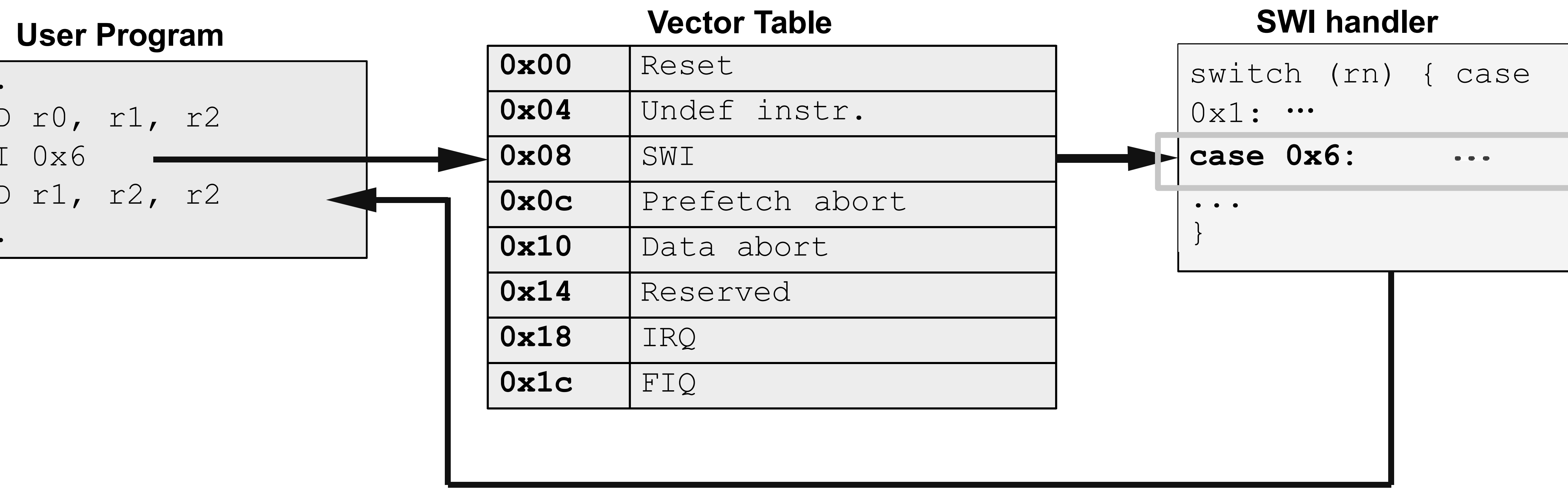
Processor Actions for SWI (1)

- Save the address of the instruction after the SWI in **r14_svc**
- Save the **CPSR** in **SPSR_svc**
- Enter supervisor mode
- Disable IRQs
- Set the **PC** to 0x8

Processor Actions for SWI (2)



Processor Actions for SWI (3)



```
.
D r0, r1, r2
I 0x6
D r1, r2, r2
.
```

0x00	Reset
0x04	Undef instr.
0x08	SWI
0x0c	Prefetch abort
0x10	Data abort
0x14	Reserved
0x18	IRQ
0x1c	FIQ

```
switch (rn) { case
0x1: …
case 0x6:     …
...
}
```

ARM Instruction Set

- Data processing instructions
- Data transfer instructions
- Control flow instructions
- **Writing simple assembly language programs**

Writing Simple Assembly Language Programs

(ARM ADS)

```
     AREA      HelloW, CODE, READONLY
I_WriteC                 EQU         &0
I_Exit                   EQU         &11

           ENTRY
ART    ADR               r1, TEXT
OP         LDRB          r0, [r1], #1
           CMP           r0, #0
           SWINE    SWI_WriteC BNE
                    LOOP
           SWI      SWI_Exit
XT         =             "Hello World",&0a,&0d,0
           END
```

AREA: chunks of data or code that are manipulated by the linker

EQU: give a symbolic name to a numeric constant (*)

DCB: allocate one or more bytes of memory and define initial runtime content of memory (=)

TRY: The first instruction to be executed within an application is marked by the ENTRY ective. An application can contain only a single entry point.

General Assembly Form (ARM ADS)

```
label <whitespace> instruction <whitespace> ;comment
```

- The three sections are separated by at least one whitespace character (a space or a tab)
- Actual instructions never start in the first column, since they must be preceded by whitespace, even if there is no label
- All three sections are optional

GNU GAS Basic Format (1)

```
        .section .text
        .global main
        .type main,%function
main:
         MOV r0, #100
         ADD r0, r0, r0
        .end
```

Filename: test.s

- Assemble the following code into a section
- Similar to “AREA” in armasm

GNU GAS Basic Format (2)

```
        .section .text
        .global main
        .type main,%function
main:
         MOV r0, #100
         ADD r0, r0, r0
        .end
```

Filename: test.s

- ".global" makes the symbol visible to ld
- Similar to "EXPORT" in armasm

GNU ARM Basic Format (3)

```
        .section .text
        .global main
        .type main,%function
main:
         MOV r0, #100
         ADD r0, r0, r0
        .end
```

Filename: test.s

- This sets the type of symbol name to be either a function symbol or an objec symbol

- “.end” marks the end of the assembly file
- Assembler does not process anything in the file past the “.end” directive

GNU ARM Basic Format (4)

```
        .section .text
        .global main
        .type main,%function
main:
         MOV r0, #100
         ADD r0, r0, r0
        .end
```

Filename: test.s

- LABEL
- armasm

- Comments
  - /* …your comments... */
  - @ your comments   (line comment)

Thumb Instruction Set

- **Thumb addresses code density**
  - A compressed form of a subset of the ARM instruction set
- **Thumb maps onto ARMs**
  - Dynamic decompression in an ARM instruction pipeline
  - Instructions execute as standard ARM instructions within the processor
- **Thumb is not a complete architecture**
- **Thumb is fully supported by ARM development tools**
- **Design for processor / compiler, not for programmer**

Thumb-ARM Differences (1)

- All Thumb instructions are 16-bits long
  - ARM instructions are 32-bits long
- Most Thumb instructions are executed **unconditionally**
  - All ARM instructions are executed conditionally

Thumb-ARM Differences (2)

- Many Thumb data processing instructions use a **2-address** format (the destination register is the same as one of the source registers)
  - ARM use 3-address format
- Thumb instruction are less regular than ARM instruction formats, as a result of the dense encoding

Thumb Applications

- **Thumb properties**
  - Thumb requires **70%** space of the ARM code
  - Thumb uses **40%** more instructions than the ARM code
  - With 32-bit memory, the ARM code is **40%** faster than the Thumb code
  - With 16-bit memory, the Thumb code is **45%** faster than the ARM code
  - Thumb uses **30%** less external memory power than ARM code

END