

Introduction to ARM Processors

OUTLINE

-Background

-ARM Microprocessor

- ARM Architecture,
- Assembly Language Programming
- Instruction Set

BACKGROUND

- Architectural features of embedded processor
- **General rules (with exceptions):**
 1. Designed for efficiency (vs. ease of programming)
 2. Huge variety of processors (resulting from 1.)
 3. Harvard architecture
 4. Heterogeneous register sets
 5. Limited instruction-level parallelism or VLIW ISA
 6. Different operation modes (saturating arithmetic, fixed point)
 7. Specialised microcontroller & DSP instructions (bit-field addressing, multiply/accumulate, bit-reversal, modulo addressing)
 8. Multiple memory banks
- 9. No “fat” (MMU, caches, memory protection, target buffers, complex pipeline logic, ...)
- **These features have to be known to the compiler!**

ARM Concept

- What is ARM?

- Advanced RISC Machine
- Acorn and VLSI Technology built in 1990/11
- RISC
- IP Core
- T.I. , PHILIPS , INTEL.....
- RISC Microcontroller
 - ARM7、ARM9、ARM9E-S、StrongARM
ARM10.....

ARM的產品是 IP Core, 業務是銷售晶片系統的核心技術IP, 全球有許多大型IT公司採用ARM的技術, 如TI, Intel。

ARM的專利收入主要來自專利授權金以及按比例收取產品的專利使用費

ARM Concept

- Why ARM?
 - Low power 、 Low cost 、 Tiny
 - 8/16/32 bit microprocessor
 - Thumb mode
 - Namely
 - T : Thumb Mode
 - D : Debug interface (JTAG)
 - M : Multiplier
 - I : ICE interface (Trace 、 Break point)

Why ARM here?

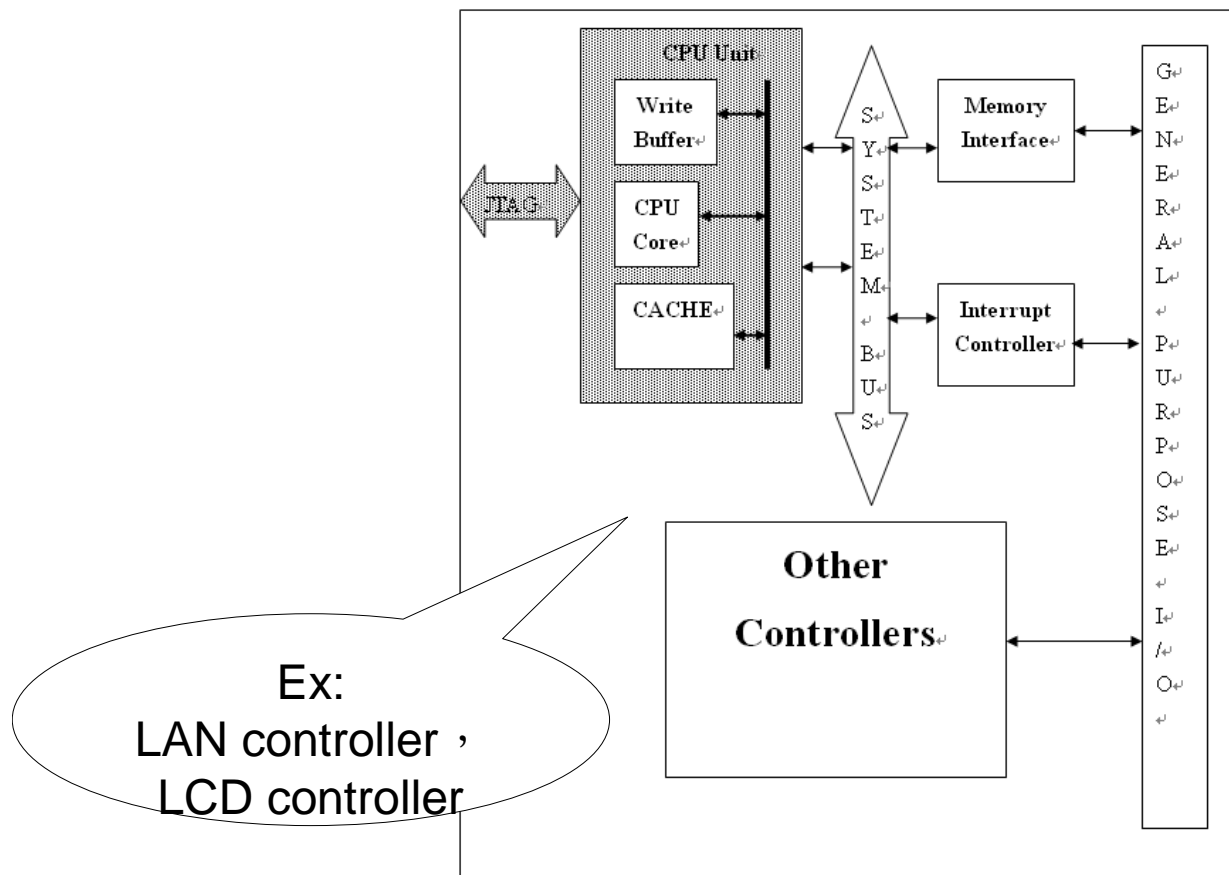
- ARM is one of the most licensed and thus widespread processor cores in the world
- Used especially in portable devices due to low power consumption and reasonable performance (MIPS / watt)
- Several interesting extensions available or in development like Thumb instruction set and Jazelle Java machine
 - <http://www.arm.com/armtech/jazelle?OpenDocument>

ARM processor

- ARM is a family of RISC architectures.
- “ARM” is the abbreviation of “Advanced RISC Machines”.
- ARM does not manufacture its own VLSI devices.
 - lincses
- ARM7- von Neuman Architecture
- ARM9 – Harvard Architecture

ARM vs. SoC

- Architecture of ARM and SoC



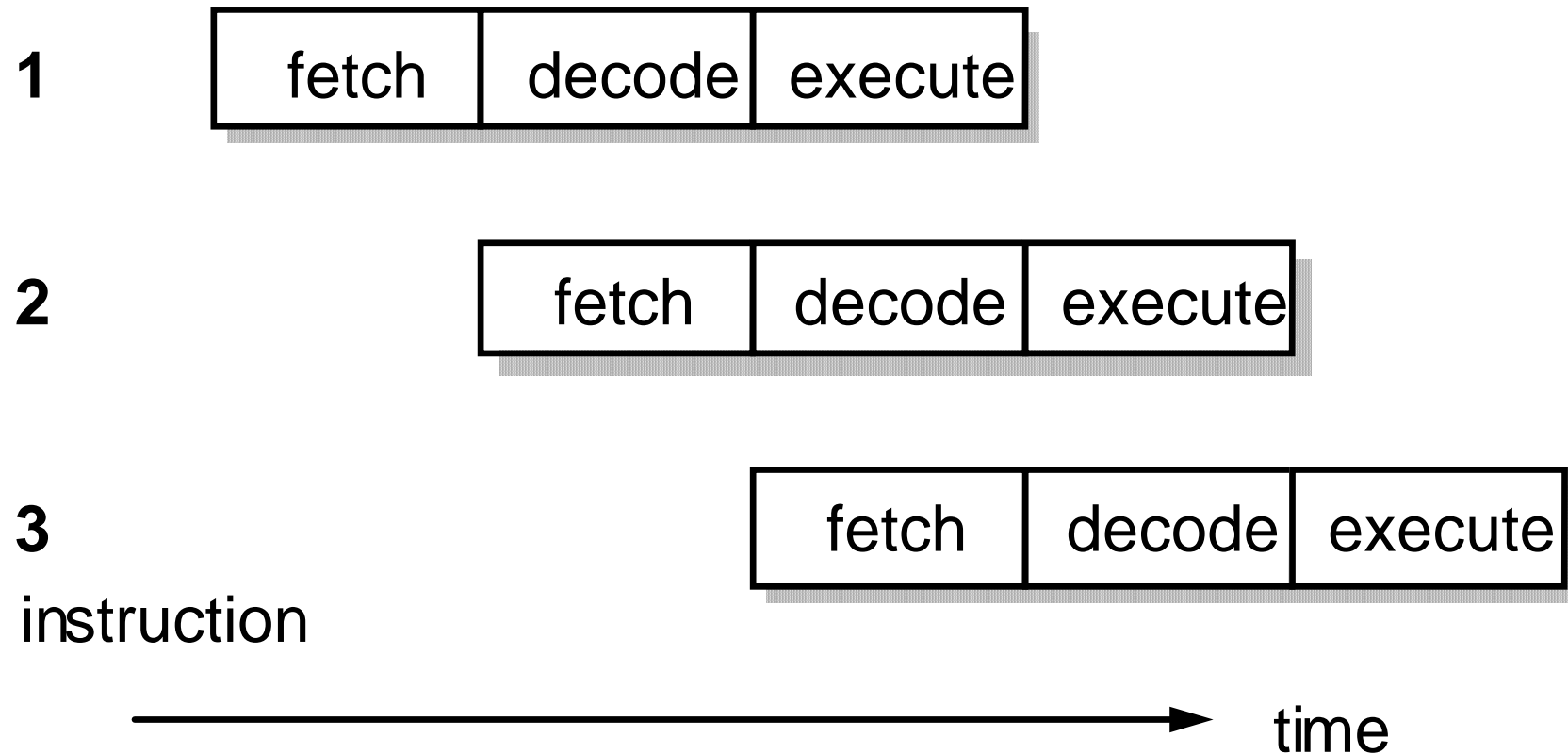
ARM核心就是個CPU，
SoC則是把系統要的功能全放到CPU內，可以提供特定用途的單晶片IC。以個人電腦為例，將一部電腦除了電源外，皆轉變到一顆IC中。

	Cache Size (Inst/Data)	Tightly Coupled Memory	Memory Manage- ment	AHB Bus Interface	Thumb	DSP	Jazelle	Clock MHz **
Embedded Cores								
<u>ARM7TDMI</u>	No	No	No	Yes*	Yes	No	No	133
<u>ARM7TDMI-S</u>	No	No	No	Yes*	Yes	No	No	100-133
<u>ARM7EJ-S</u>	No	No	No	Yes*	Yes	Yes	Yes	100-133
<u>ARM966E-S</u>	No	Yes	No	Yes	Yes	Yes	No	230-250
<u>ARM940T</u>	4K/4K	No	MPU	Yes*	Yes	No	No	180
<u>ARM946E-S</u>	Variable	Yes	MPU	Yes	Yes	Yes	No	180-210
<u>ARM1026EJ-S</u>	Variable	Yes	MMU+MPU	dual AHB	Yes	Yes	Yes	266-325
Platform Cores								
<u>ARM720T</u>	8K unified	No	MMU	Yes	Yes	No	No	100
<u>ARM920T</u>	16K/16K	No	MMU	Yes*	Yes	No	No	250
<u>ARM922T</u>	8K/8K	No	MMU	Yes*	Yes	No	No	250
<u>ARM926EJ-S</u>	Variable	Yes	MMU	dual AHB	Yes	Yes	Yes	220-250
<u>ARM1020E</u>	32K/32K	No	MMU	dual AHB	Yes	Yes	No	325
<u>ARM1022E</u>	16K/16K	No	MMU	dual AHB	Yes	Yes	No	325
<u>ARM1026EJ-S</u>	Variable	Yes	MMU+MPU	dual AHB	Yes	Yes	Yes	266-325
Secure Applications								
<u>SC100</u>	No	No	MPU	No	Yes	No	No	80
<u>SC110</u>	No	No	MPU	No	Yes	No	No	80
<u>SC200</u>	Optional	No	MPU	No	Yes	Yes	Yes	110
<u>SC210</u>	Optional	No	MPU	No	Yes	Yes	Yes	110
Intel ARM-based Processors								
<u>StrongARM</u>	16K/8K	No	MMU	N/A	No	No	No	206
<u>Intel XScale</u>	32K/32K	No	MMU	N/A	Yes	Yes	No	400

Intel Xscale

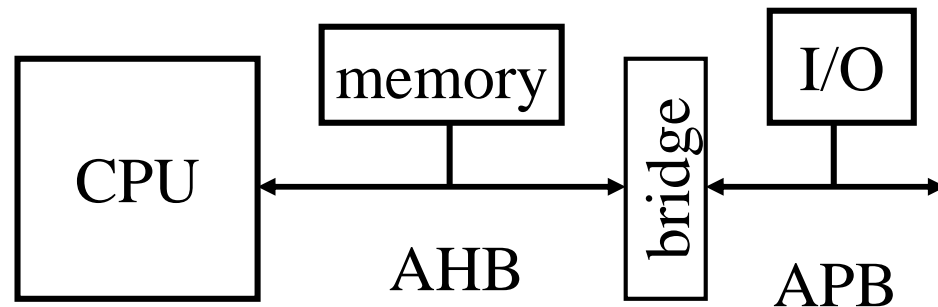
- ARM* Architecture Version 5TE ISA compliant.
 - ARM* Thumb Instruction Support
 - ARM* DSP Enhanced Instructions
- Low power consumption and high performance
- Intel® Media Processing Technology
 - Enhanced 16-bit Multiply
 - 40-bit Accumulator
- 32-KByte Instruction Cache
- 32-KByte Data Cache
- 2-KByte Mini Data Cache
- 2-KByte Mini Instruction Cache
- Instruction and Data Memory Management Units
- Branch Target Buffer
- Debug Capability via JTAG Port

ARM single-cycle instruction 3-stage pipeline operation



ARM busses

- AMBA:
 - Open standard.
 - Many external devices.
- Two varieties:
 - AMBA High-Performance Bus (AHB).
 - AMBA Peripherals Bus (APB).



ARM instruction set

- ARM processor (operating) states
- ARM memory organization.
- ARM programming model.
- ARM assembly language.
- ARM data operations.
- ARM flow of control.
- C to assembly examples
- Exceptions
- Coprocessor instructions
- Summary

Processor Operating States

- The ARM7TDMI processor has two operating states:
 - ARM - 32-bit, word-aligned ARM instructions are executed in this state.
 - Thumb -16-bit, halfword-aligned Thumb instructions are executed in this state.

- The operating state of the ARM7TDMI core can be switched between ARM state and Thumb state using the BX (branch and exchange) instructions

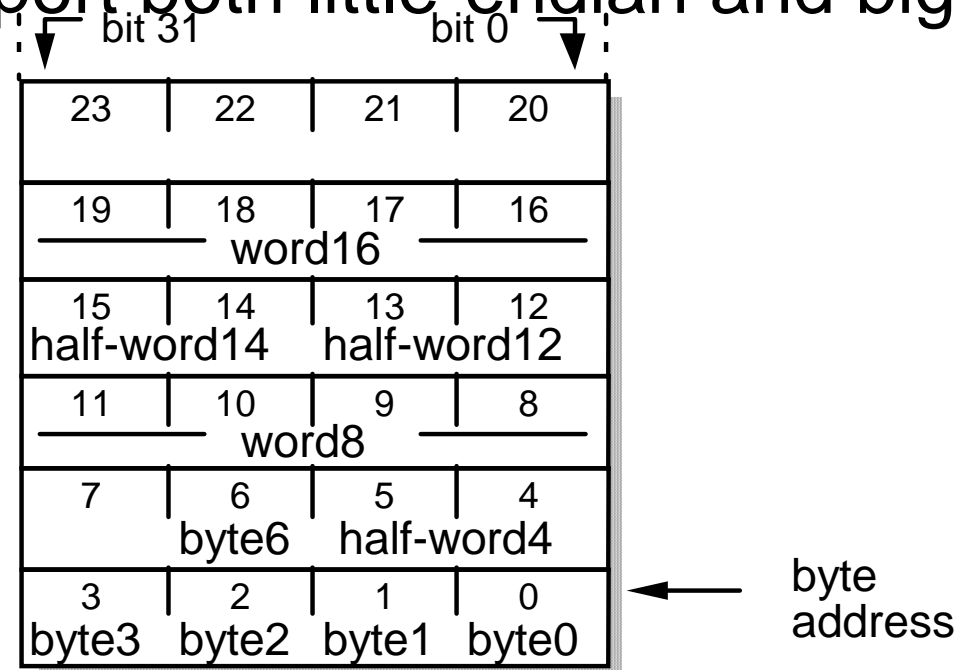
`BX{<cond>} <Rm>`

where:

<code><cond></code>	Is the condition under which the instruction is executed. The conditions are defined in <i>The condition field</i> on page A3-5. If <code><cond></code> is omitted, the AL (always) condition is used.
<code><Rm></code>	Holds the value of the branch target address. Bit[0] of Rm is 0 to select a target ARM instruction, or 1 to select a target Thumb instruction.

The Memory System

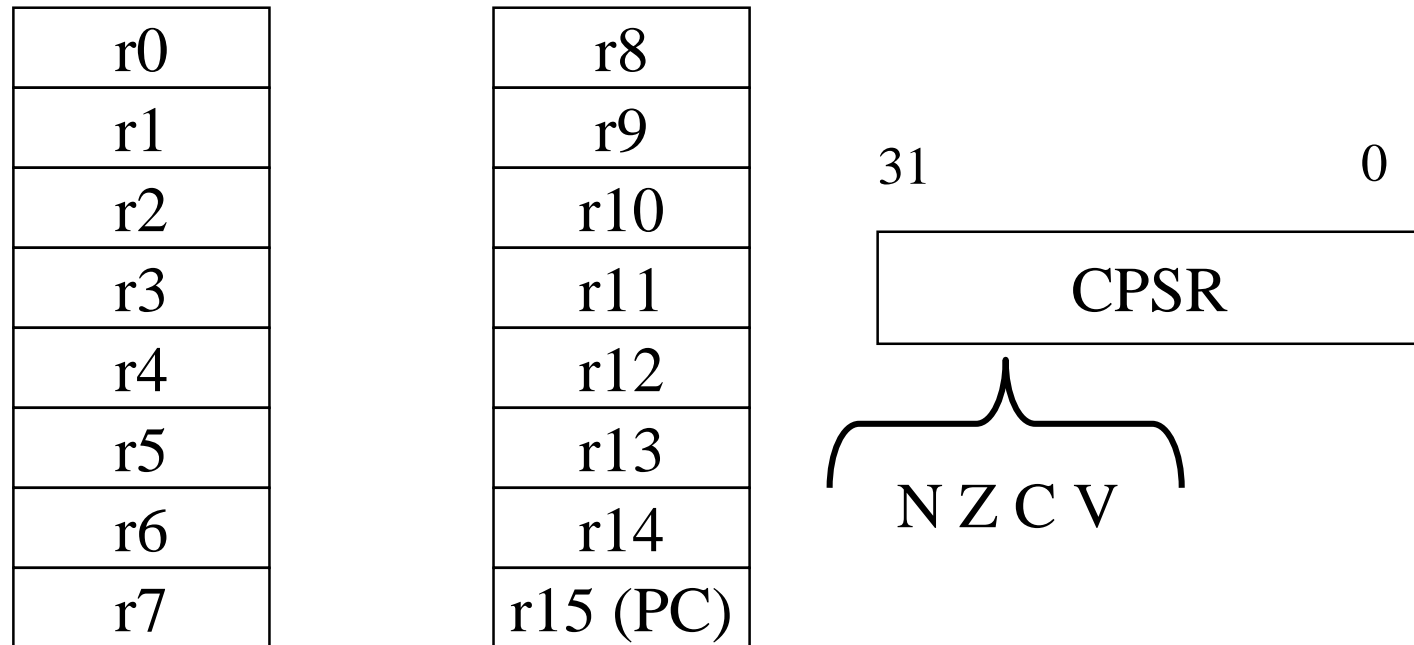
- 4 G address space
 - 8-bit bytes, 16-bit half-words, 32-bit words
 - Support both little-endian and big-endian



Operating Modes

- The ARM7TDMI processor has seven modes of operations:
 - User mode(usr)
 - Normal program execution mode
 - Fast Interrupt mode(fiq)
 - Supports a high-speed data transfer or channel process.
 - Interrupt mode(irq)
 - Used for general-purpose interrupt handling.
 - Supervisor mode(svc)
 - Protected mode for the operating system.
 - Abort mode(abt)
 - implements virtual memory and/or memory protection
 - System mode(sys)
 - A privileged user mode for the operating system. (runs OS tasks)
 - Undefined mode(und)
 - supports a software emulation of hardware coprocessors
- Except user mode, all are known as privileged mode.

ARM programming model

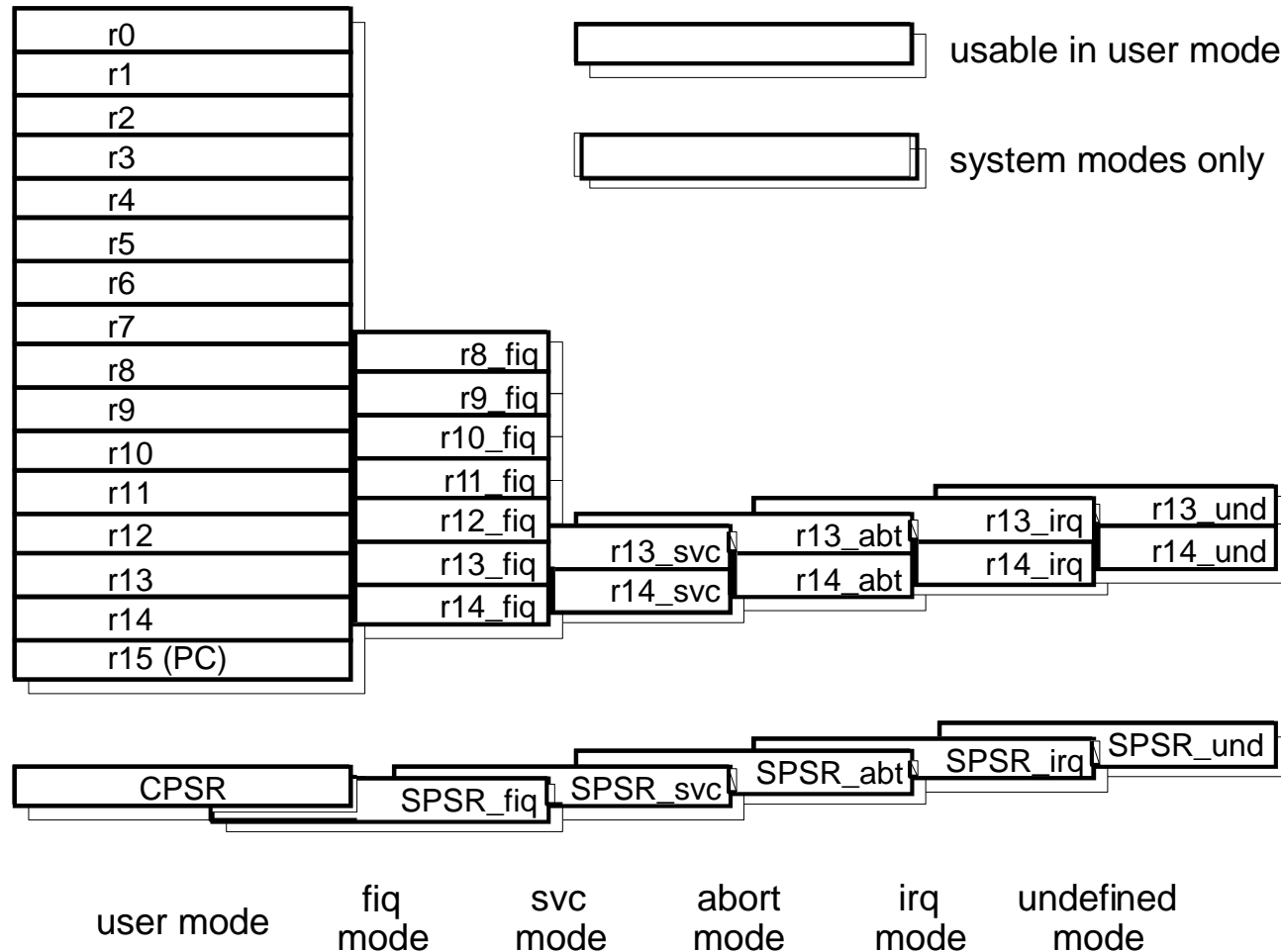


CPSR: Current Program Status Register
SPSR: Saved Program Status Register

Registers

- 37 registers
 - 31 general 32 bit registers, including PC
 - 6 status registers
 - 15 general registers (R0 to R14), and one status registers and program counter are visible at any time – when you write user-level programs
 - R13 (SP)
 - R14 (LR)
 - R15 (PC)
- The visible registers depend on the processor mode
- The other registers (the banked registers) are switched in to support IRQ, FIQ, Supervisor, Abort and Undefined mode processing

ARM Registers (1)



Registers

- R0 to R15 are directly accessible
- R0 to R14 are general purpose
- R13: Stack point (sp) (in common)
 - Individual stack for each processor mode
- R14: Linked register (lr)
- R15 holds the Program Counter (PC)
- CPSR - Current Program Status Register contains condition code flags and the current mode bits
- 5 SPSRs (Saved Program Status Registers) which are loaded with CPSR when an exceptions occurs

The Program Counter (R15)

- When the processor is executing in ARM state:
 - All instructions are 32 bits in length
 - All instructions must be word aligned
 - Therefore the PC value is stored in bits [31:2] with bits [1:0] equal to zero (as instruction cannot be halfword or byte aligned).
- R14 is used as the subroutine link register (LR) and stores the return address when Branch with Link (BL) operations are performed, calculated from the PC.
- Thus to return from a linked branch

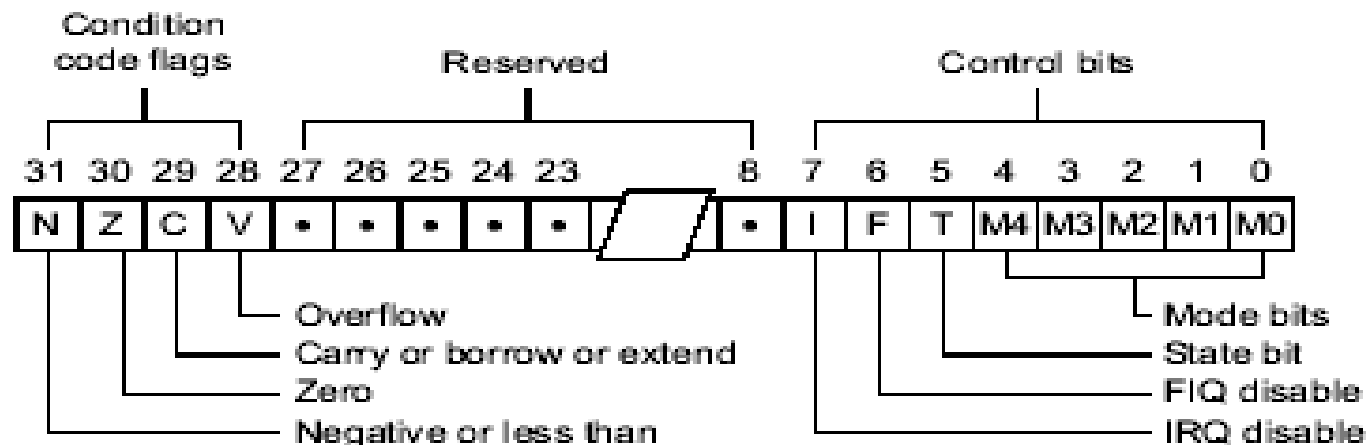
```
MOV r15,r14
MOV pc,lr
```

Program Status Registers

- The ARM contains a Current Program Status Register (CPSR), plus five Saved Program Status Registers (SPSRs) for use by exception handlers.
- These register's functions are:
 - Hold information about the most recently performed ALU operation.
 - Control the enabling and disabling of interrupts.
 - Set the processor operating mode

Program Status Registers

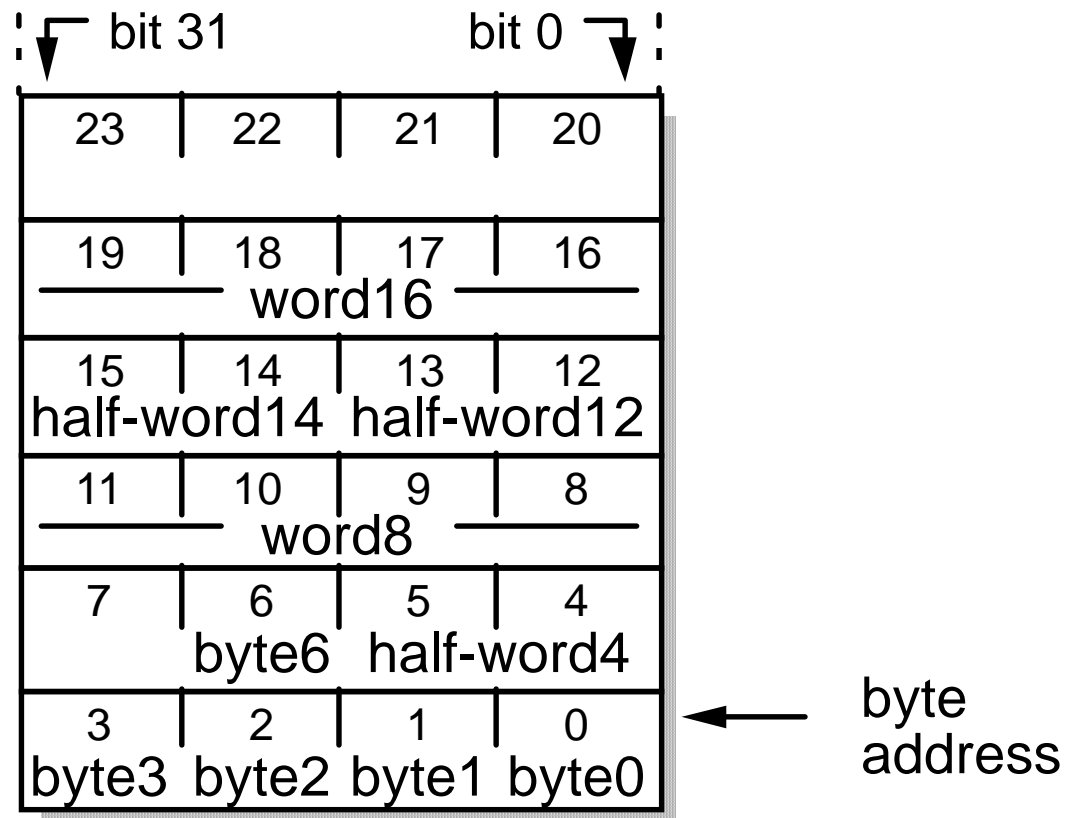
- The N, Z, C and V are condition code flags
 - may be changed as a result of arithmetic and logical operations in the processor
 - may be tested by all instructions to determine if the instruction is to be executed
 - N : Negative. Z : Zero. C : Carry. V : oVerflow
- The I and F bits are the interrupt disable bits
- The T bit is thumb bit
- The M0. M1. M2. M3 and M4 bits are the mode bits



Program Counter (r15)


- When the processor is executing in ARM state:
 - All instructions are **32 bits wide**
 - All instructions must be **word aligned**
 - The **PC** value is stored in bits [31:2] with bits [1:0] undefined
 - Instructions cannot be halfword or byte aligned

ARM Memory Organization




Big Endian and Little Endian

Big endian

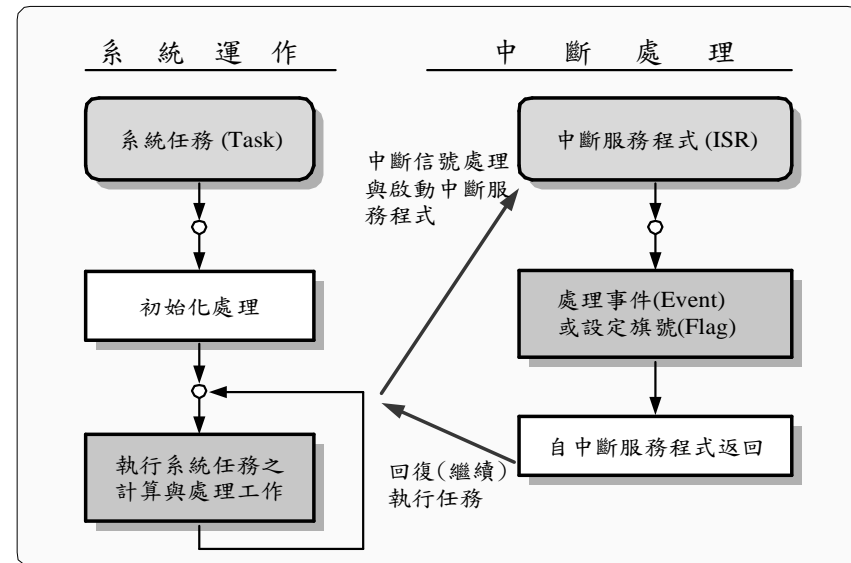
Higher Address	31	24	23	16	15	8	7	0	Word Address
	8		9		10		11		8
	4		5		6		7		4
	0		1		2		3		0
Lower Address	<ul style="list-style-type: none">• Most significant byte is at lowest address• Word is addressed by byte address of most significant byte								

Little endian

Higher Address	31	24	23	16	15	8	7	0	Word Address
	11		10		9		8		8
	7		6		5		4		4
	3		2		1		0		0
Lower Address	<ul style="list-style-type: none">• Least significant byte is at lowest address• Word is addressed by byte address of least significant byte								

Exceptions

- **Exceptions are usually used to handle unexpected events which arise during the execution of a program**



From 黃悅民等嵌入式系統設計-以ARM 處理器為基礎之 SoC平台

Exception

- **System Exception**
 - CPU在執行時，愈到特殊的狀況而產生的例外，使用者完全無法對例外進行初始化、停止、或啓動
- **Interrupt Exception**
 - ARM CPU預留給系統建置者使用的中斷入口

Exception Groups

- **Direct effect of executing an instruction**
 - SWI
 - Undefined instructions
 - Prefetch aborts (memory fault occurring during fetch)
- **A side-effect of an instruction**
 - Data abort (a memory fault during a load or store data access)
- **Exceptions generated externally**
 - Reset
 - IRQ
 - FIQ

Exception Entry

- Change to the corresponding mode
- Save the address of the instruction following the exception instruction in **r14 of the new mode**
- Save the old value of CPSR in the **SPSR of the new mode**
- Disable IRQ
- If the exception is a FIQ, disables further FIQ
- Force PC to execute at the relevant vector address

Exception Vector Addresses

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


◆ Intel x86 – 0x00000 ~ 0x003FF (4 x 256)

◆ ARM – 0x000000 ~ 0x00001F

Exception Return

- Any modified user registers must be restored
- Restore CPSR
- Resume PC in the correct instruction stream

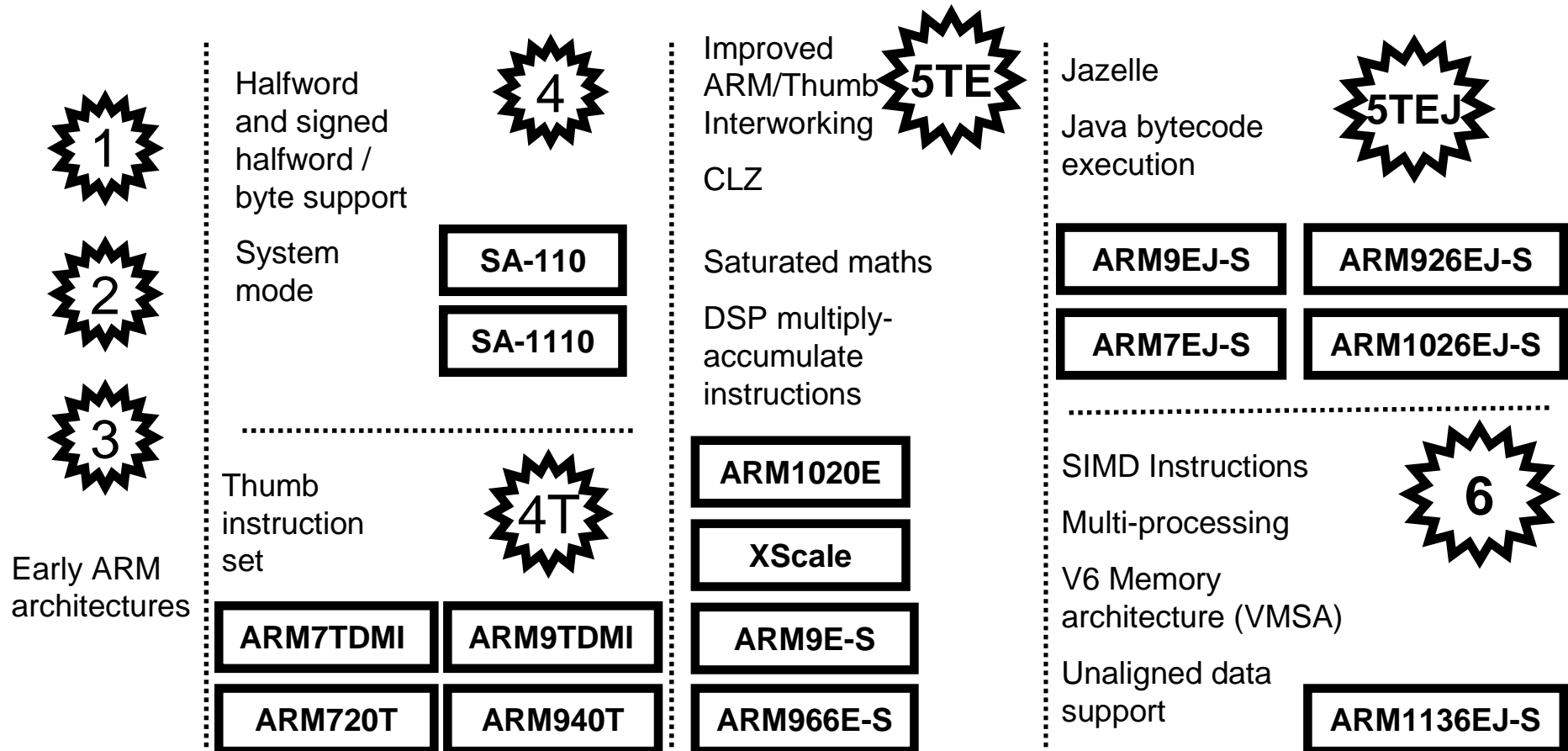
Exception Priorities

- Reset
 - Data abort
 - FIQ
 - IRQ
 - Prefetch abort
 - SWI, undefined instruction
- Highest priority
- 

Naming Rule of ARM

- **ARM {x} {y} {z} {T} {D} {M} {I} {E} {J} {F} {-S}**
 - x: series
 - y: memory management / protection unit
 - z: cache
 - T: Thumb decoder
 - D: JTAG debugger
 - M: fast multiplier
 - I: support hardware debug
 - E: enhance instructions (based on TDMI)
 - J: Jazelle
 - F: vector floating point unit
 - S: synthesiable, suitable for EDA tools

Development of the ARM Architecture



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

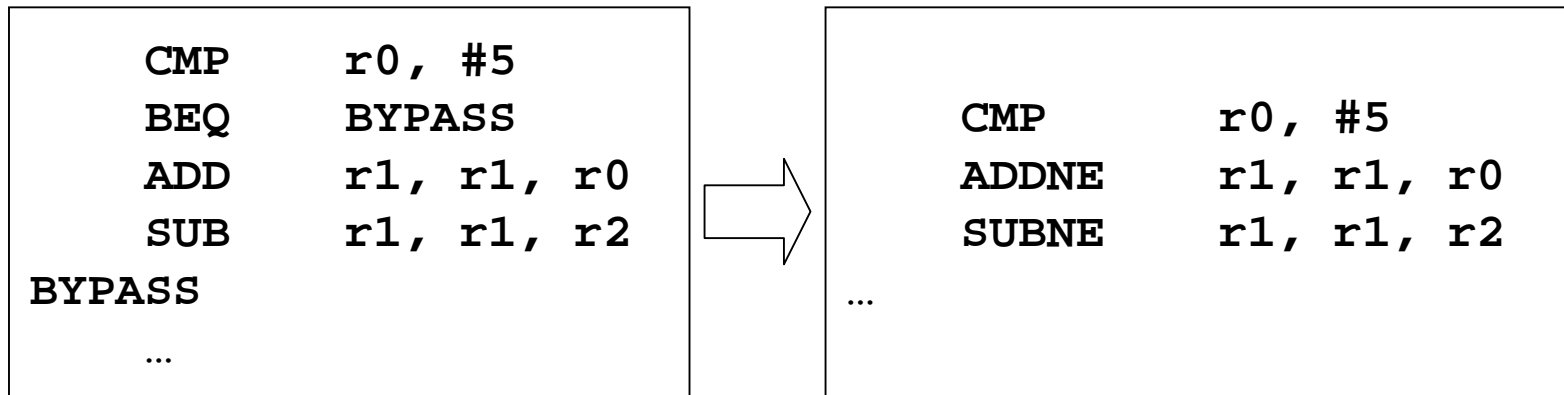
- The ARM processor is very easy to program at the assembly level
- In this part, we will
 - **Look at ARM instruction set and assembly language programming at the user level**

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



Conditional Execution (2)

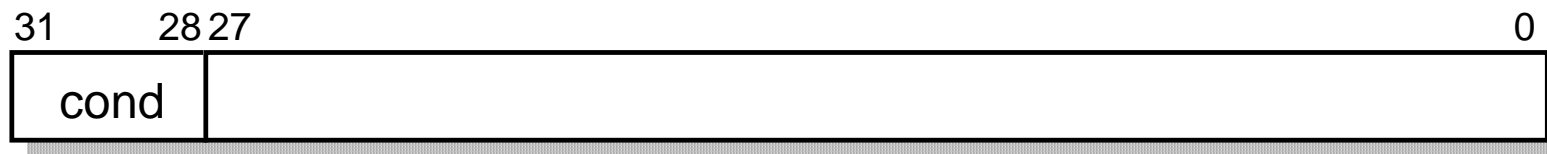
- The conditional execution code is faster and smaller

```
; 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  
    CMPEQ   r2, r3  
    ADDEQ   r4, r4, #1
```

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

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	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

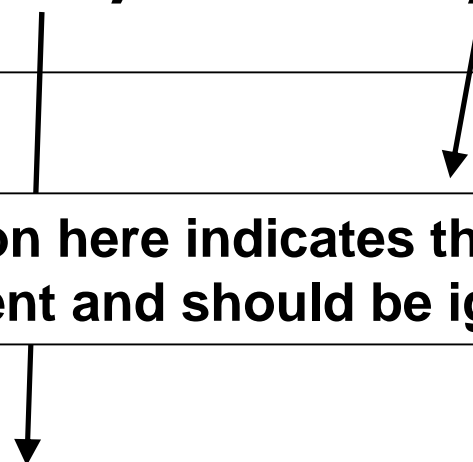
- **Data processing instructions**
- Data transfer instructions
- Control flow instructions
- Writing simple assembly language programs

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

$r0[i] := r1[i] \text{ OP}_{\text{logic}} r2[i] \quad \text{for } i \text{ in } [0..31]$
--

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 $r1 \text{ AND } r2$
TEQ	r1, r2	test equal	set cc on $r1 \text{ XOR } r2$

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]
```



The diagram shows two arrows originating from the immediate values in the assembly code. One arrow points from '#1' in the 'ADD' instruction to a box explaining that a constant is preceded by '#'. The other arrow points from '#&ff' in the 'AND' instruction to a box explaining that a hexadecimal value is indicated by an '&' after the '#'. A curved arrow also points from the '#' in '#1' to the same box.

A constant preceded by ‘#’

A hexadecimal by putting ‘&’ after the ‘#’

Shifted Register Operands (1)

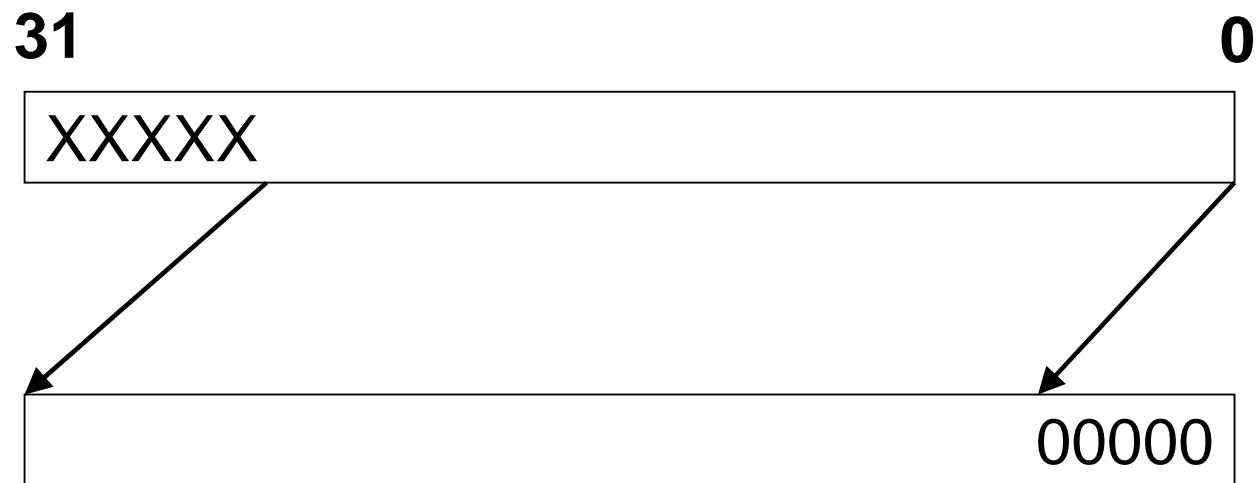
- These instructions allows **the second register operand to be subject to a shift operation** before it is combined with the first operand

```
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)

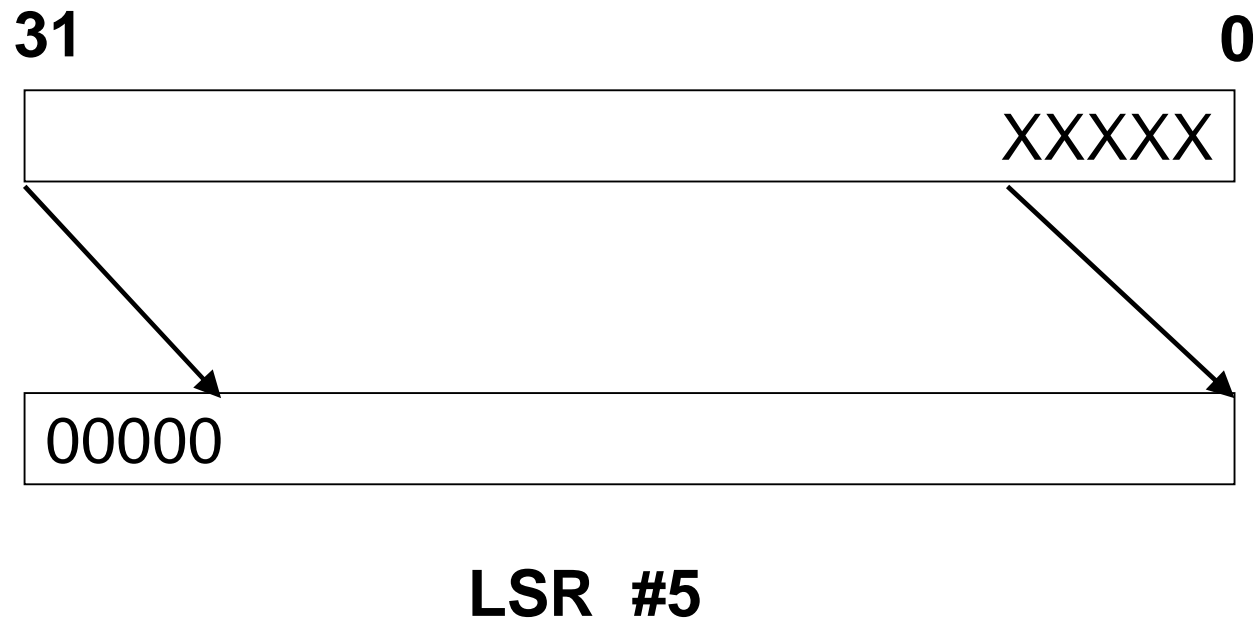
LSL	logical shift left by 0 to 31	Fill the vacated bits at the LSB of the word with zeros
ASL	arithmetic shift left	A synonym for LSL



LSL #5

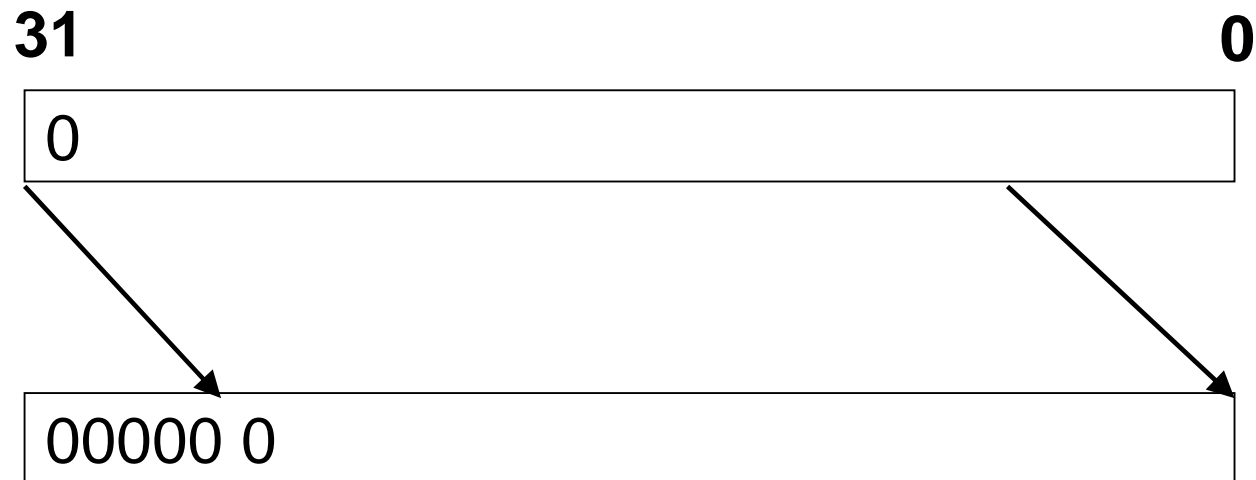
Shifted Register Operands (3)

LSR	logical shift right by 0 to 32	Fill the vacated bits at the MSB of the word with zeros
-----	--------------------------------	---



Shifted Register Operands (4)

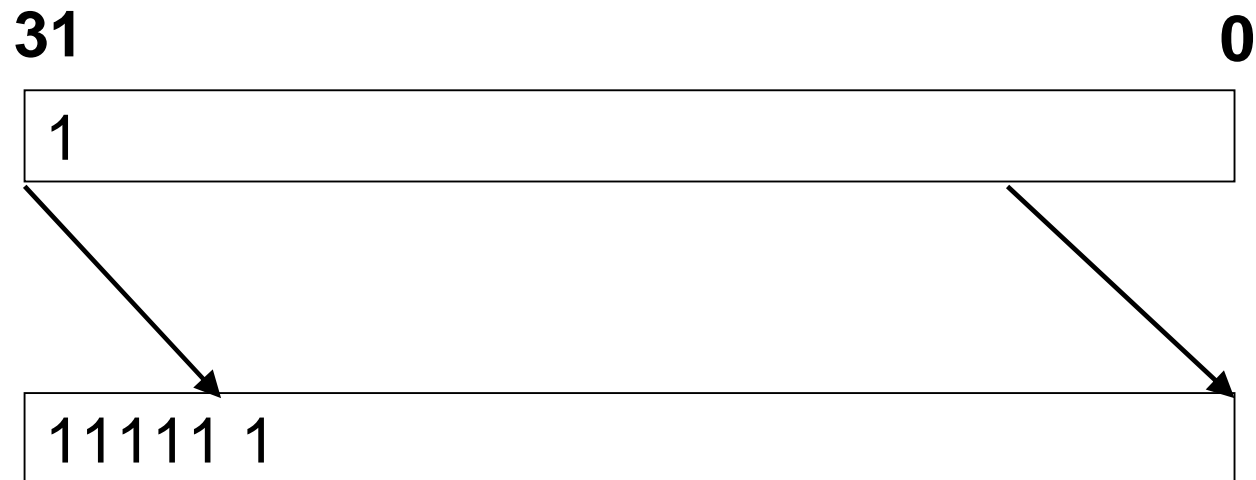
ASR	arithmetic shift right by 0 to 32	Fill the vacated bits at the MSB of the word with zero (source operand is positive)
-----	-----------------------------------	---



ASR #5 ;positive operand

Shifted Register Operands (5)

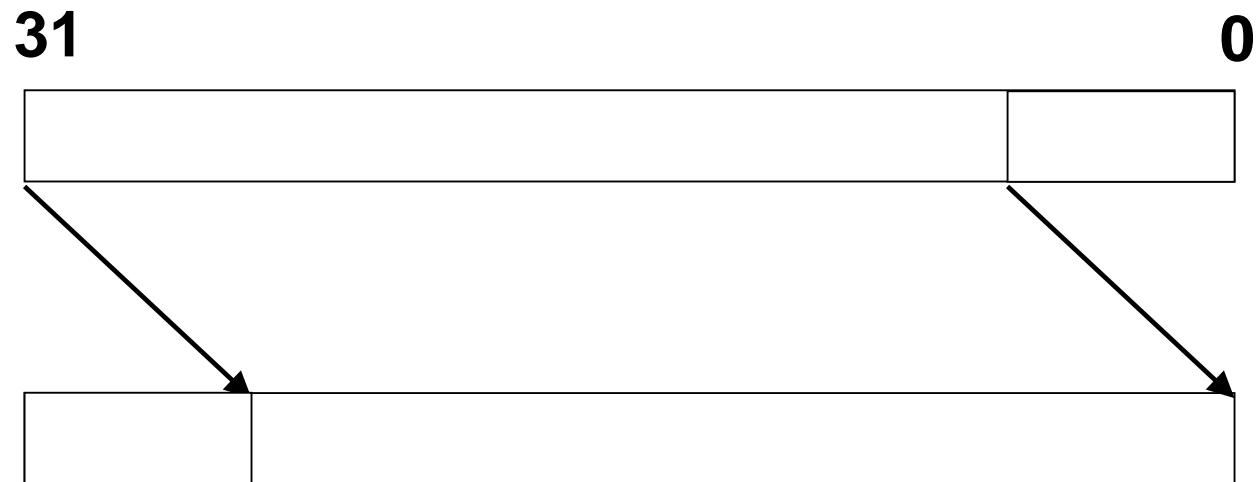
ASR	arithmetic shift right by 0 to 32	Fill the vacated bits at the MSB of the word with one (source operand is negative)
-----	-----------------------------------	--



ASR #5 ;negative operand

Shifted Register Operands (6)

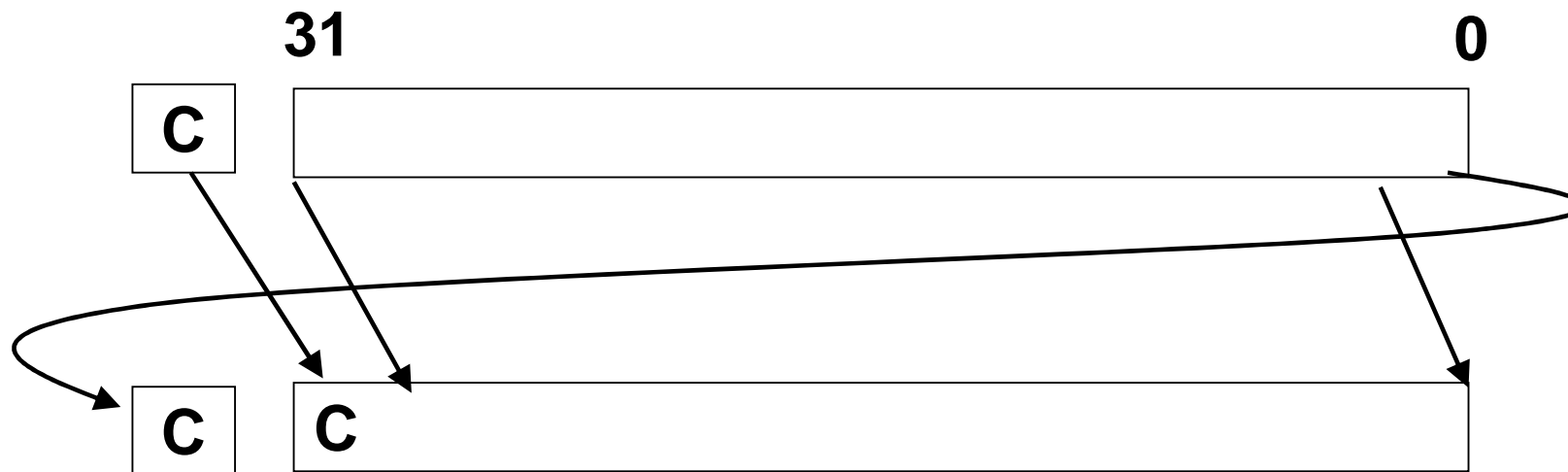
ROR	Rotate right by 0 to 32	The bits which fall off the LSB of the word are used to fill the vacated bits at the MSB of the word
-----	-------------------------	--



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

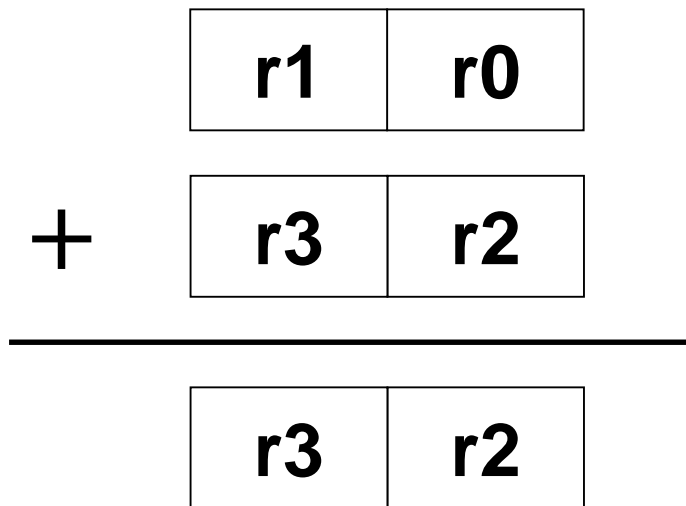
- Ex:

```
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)

LDR	Load a word into register	$Rd \leftarrow \text{mem32}[\text{address}]$
STR	Store a word in register into memory	$\text{Mem32}[\text{address}] \leftarrow Rd$
LDRB	Load a byte into register	$Rd \leftarrow \text{mem8}[\text{address}]$
STRB	Store a byte in register into memory	$\text{Mem8}[\text{address}] \leftarrow Rd$
LDRH	Load a half-word into register	$Rd \leftarrow \text{mem16}[\text{address}]$
STRH	Store a half-word in register into memory	$\text{Mem16}[\text{address}] \leftarrow Rd$
LDRSB	Load a signed byte into register	$Rd \leftarrow \text{signExtend}(\text{mem8}[\text{address}])$
LDRSH	Load a signed half-word into register	$Rd \leftarrow \text{signExtend}(\text{mem16}[\text{address}])$

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

<code>LDR</code>	<code>r0, [r1, #4]!</code>	<code>; r0 := mem₃₂[r1 + 4]</code>
		<code>; r1 := r1 + 4</code>



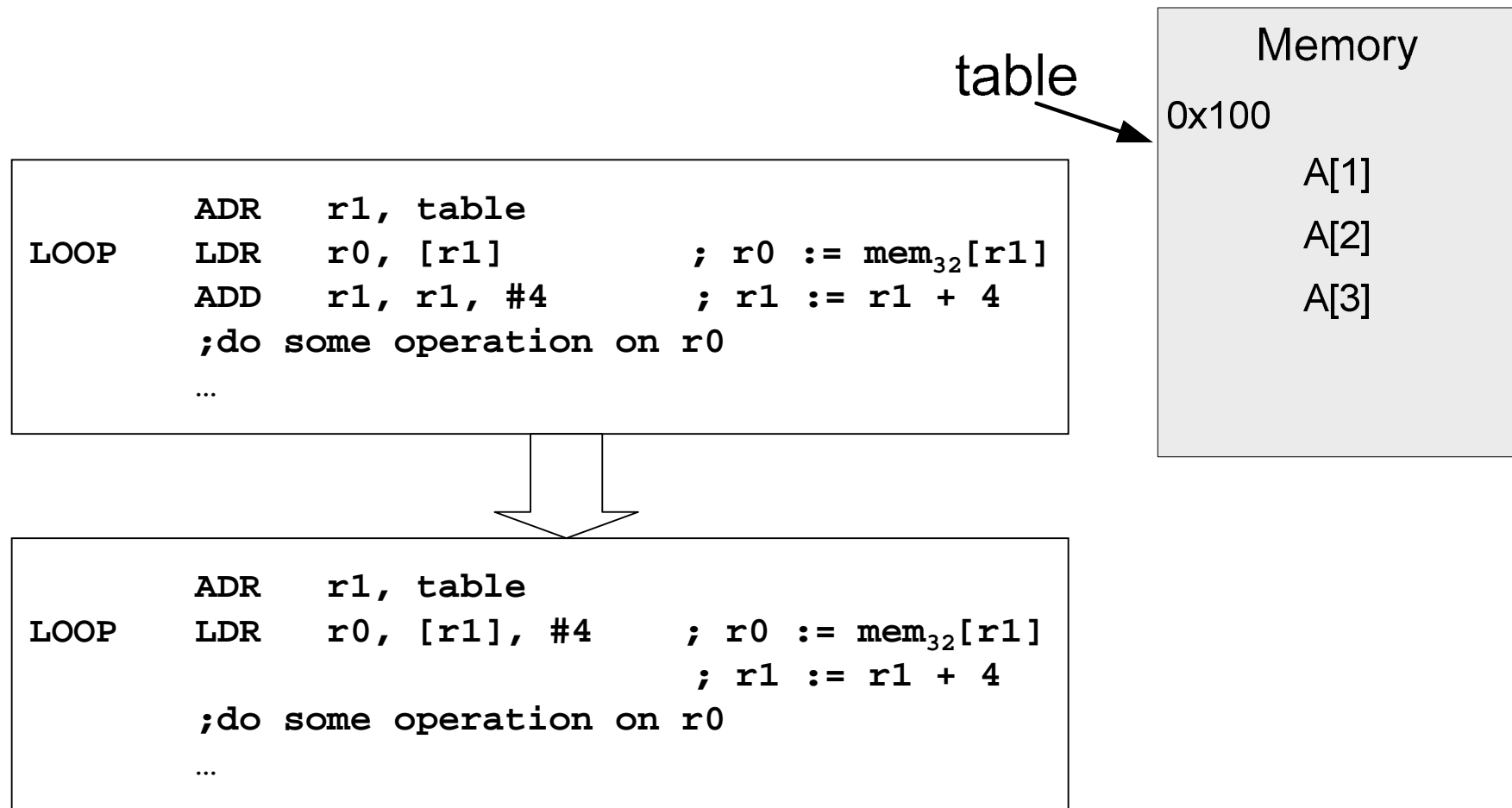
<p>The exclamation “!” mark indicates that the instruction should update the base register after initiating the data transfer</p>

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



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)

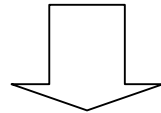
LDM	Load multiple registers
STM	Store multiple registers

Addressing mode	Description	Starting address	End address	Rn!
IA	Increment After	R_n	$R_n + 4 * N - 4$	$R_n + 4 * N$
IB	Increment Before	$R_n + 4$	$R_n + 4 * N$	$R_n + 4 * N$
DA	Decrement After	$R_n - 4 * R_n + 4$	R_n	$R_n - 4 * N$
DB	Decrement Before	$R_n - 4 * N$	$R_n - 4$	$R_n - 4 * N$

Addressing mode for multiple register load and store instructions

Example (1)

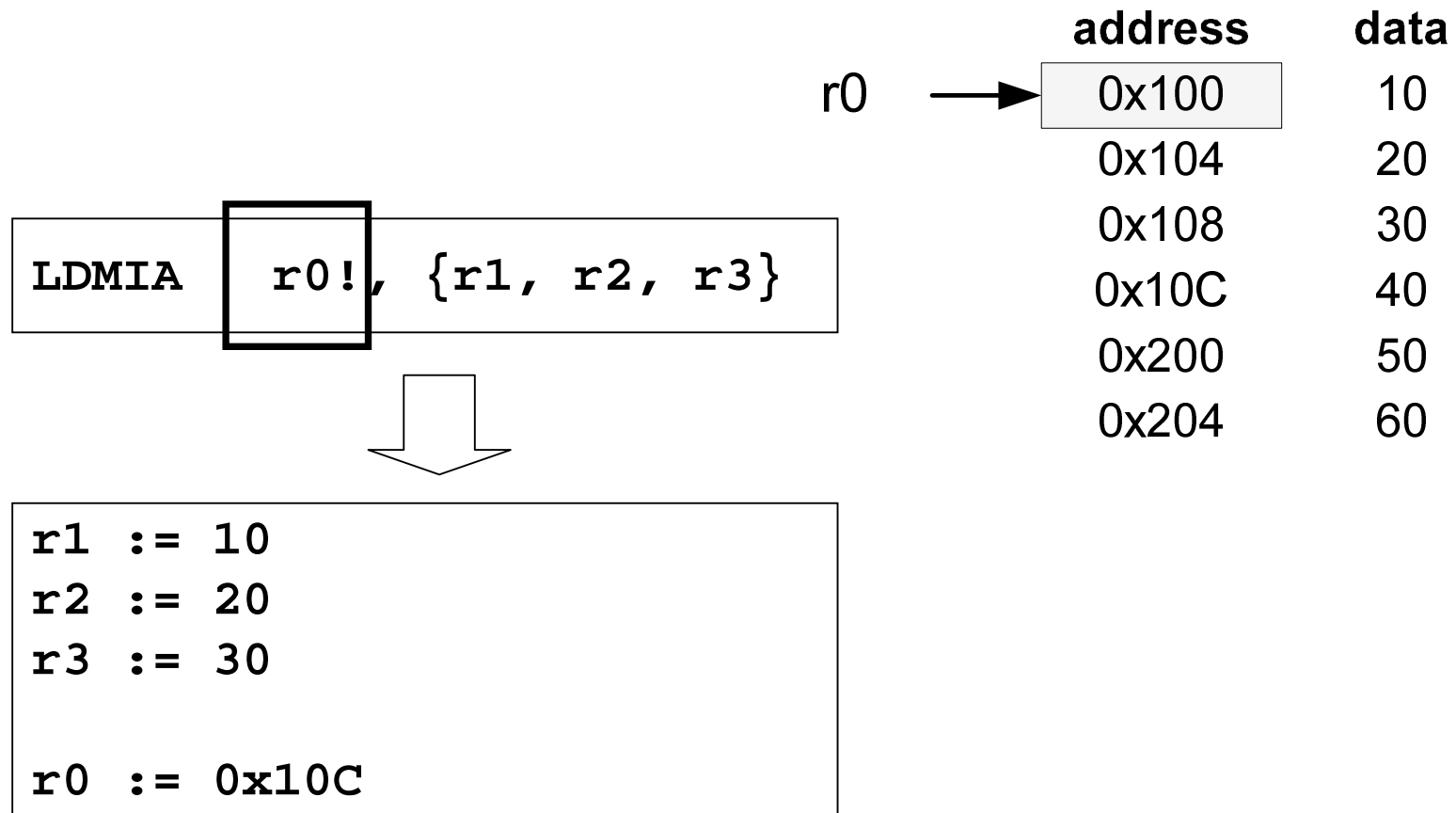
```
LDMIA    r0, {r1, r2, r3}  
OR  
LDMIA    r0, {r1-r3}
```



```
r1 := 10  
r2 := 20  
r3 := 30  
  
r0 := 0x100
```

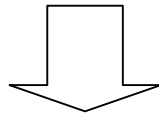
	address	data
r0 →	0x100	10
	0x104	20
	0x108	30
	0x10C	40
	0x200	50
	0x204	60

Example (2)



Example (3)

LDMIB r0!, {r1, r2, r3}



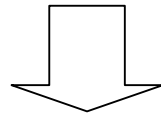
r1 := 20
r2 := 30
r3 := 40

r0 := 0x10C

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

Example (4)

LDMDA r0!, {r1, r2, r3}



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

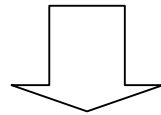
r0 := 0x108

r0 →

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

Example (5)

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



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

r0 := 0x108

r0 **→**

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

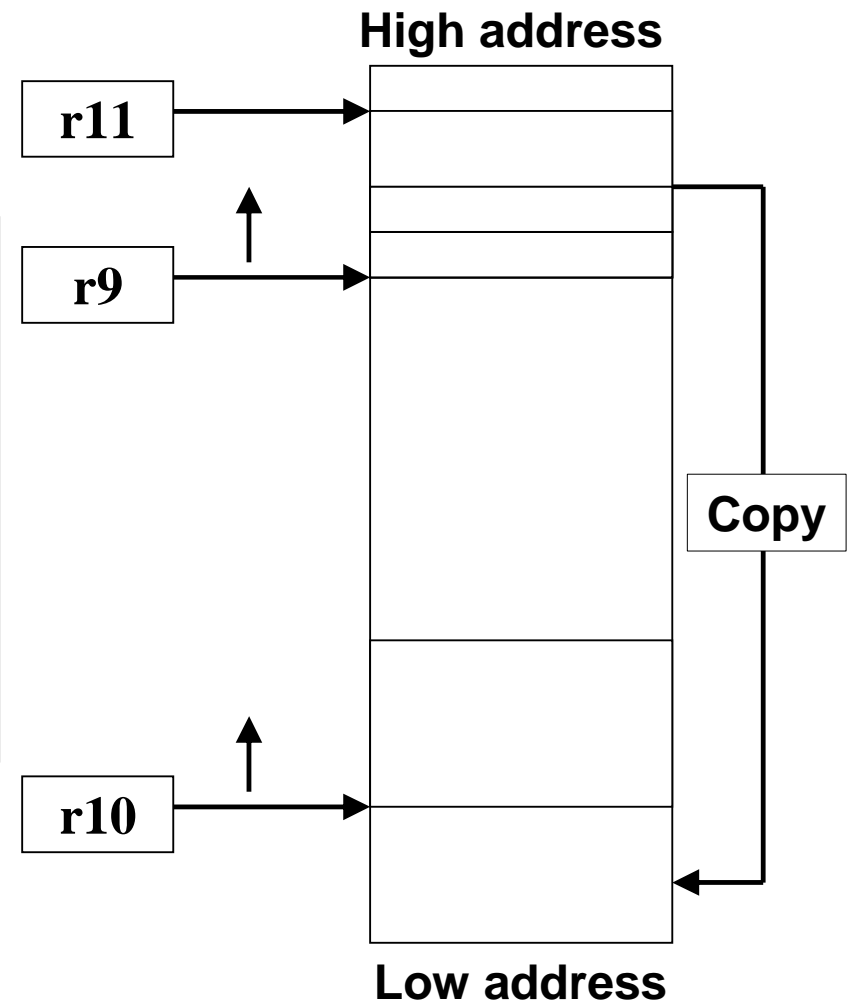
Application

Copy a block of memory

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

LOOP

```
    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	説明	POP	=LDM	PUSH	=STM
FA	遞増満	LDMFA	LFMFA	STMFA	STMIB
FD	遞減満	LDMFD	LDMIA	STMFD	STMDB
EA	遞増空	LDMEA	LDMDB	STMEA	STMIA
ED	遞減空	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)

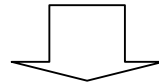
Single Register Swap Instructions (2)

SWP{B} Rd, Rm, [Rn]

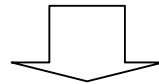
SWP	WORD exchange	tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp
SWPB	Byte exchange	tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp

Example

r0: 123456	address	data
	0x100	10
r1: 111111	0x104	20
r2: 0x108	0x108	30



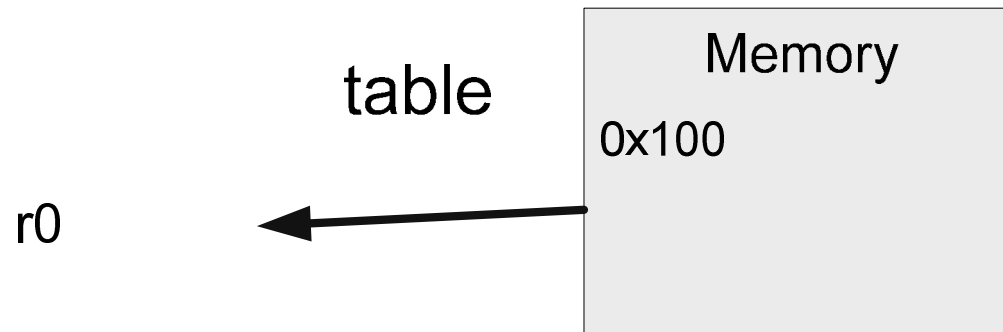
SWP r0, r1, [r2]



r0: 30	address	data
	0x100	10
r1: 111111	0x104	20
r2: 0x108	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 Debugger - E:\books\OUP3ed\ARM\arm_test

File Edit Search View Execute Options Window Help

Execution Window - arm_test.s

```

1      AREA ARMtest, CODE, READONLY
2      ENTRY
3      Start  MOV  r0,#20
4           MOV  r1,#0xFF
5           ADD  r2,r0,r1
6           ADD  r3,r0,r1, LSL #4
7           ADD  r1,r2,#65536
8
9           ADR  r5,table1
10          ADR  r6,table2
11
12      Stop  MOV  r0,#0x18
13          LDR  r1,=0x20026
14          SWI  0x123456
15
16          AREA xyz, DATA, READWRITE
17      table1 DCB "test"
18
19
20          AREA pqr, DATA, READWRITE
21      table2 DCB "test2"
22
23
24          END

```

Registers

r0	0x00000018
r1	0x00020026
r2	0x00000113
r3	0x00001004
r4	0x00000000
r5	0x000080b4
r6	0x000080ac
r7	0x00000000
r8	0x00000000
r9	0x00000000
r10	0x00000000
r11	0x00000000
r12	0x00000000
r13	0x00000000
r14	0x00000000
pc	0x000080a4
cpsr	%NZCvift_User32

Disassembly Window: 0x8080 (1)

Start	mov	r0,#0x14
0x00008084	mov	r1,#0xff
0x00008088	add	r2,r0,r1
0x0000808c	add	r3,r0,r1,lsr #4
0x00008090	add	r1,r2,#0x10000
0x00008094	add	r5,pc,#0x18
0x00008098	add	r6,pc,#0xc
Stop	mov	r0,#0x18
0x000080a0	ldr	r1,0x000080a8 ; = #0x(
0x000080a4	swi	0x123456
0x000080a8	andeq	r0,r2,r6,lsr #32
table2	ldrvcbt	r6,[r3],#-0x574
0x000080b0	andeq	r0,r0,r2,lsr r0
xyz	ldrvcbt	r6,[r3],#-0x574
_edata	andeq	r0,r0,r0

Program terminated normally

ARMulate

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	Unconditional	Always take this branch
BAL	Always	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	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	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	跳躍	PC=label
BL	帶返回的跳躍	PC=label LR=BL後面的第一道指令的位址
BX	跳躍並切換狀態	PC=Rm & 0xffffffff, T=Rm & 1
BLX	帶返回的跳躍並 切換狀態	PC=label, T=1 PC=Rm & 0xffffffff, T=Rm & 1 LR = BLX後面的第一道指令的位址

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

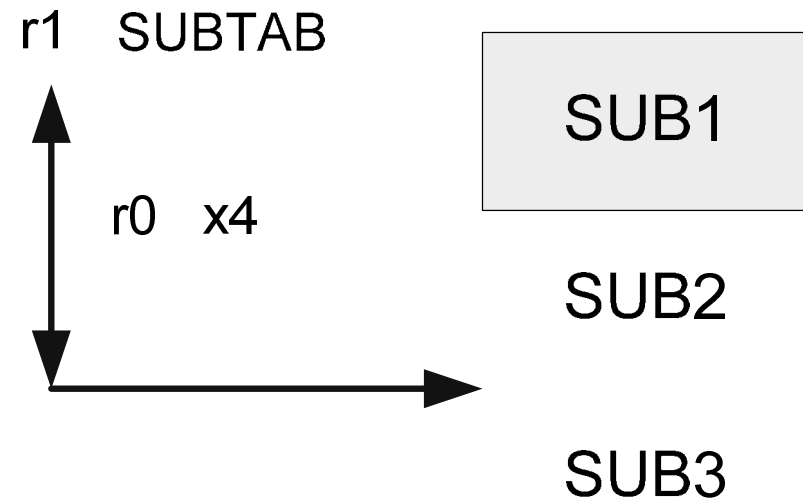
Note: slow when the list is long, and all subroutines are equally frequent

```
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
..	
JUMPTAB	
ADR	r1, SUBTAB
CMP	r0, #SUBMAX
LDRLS	pc, [r1, r0, LSL #2]
B	ERROR
SUBTAB	
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

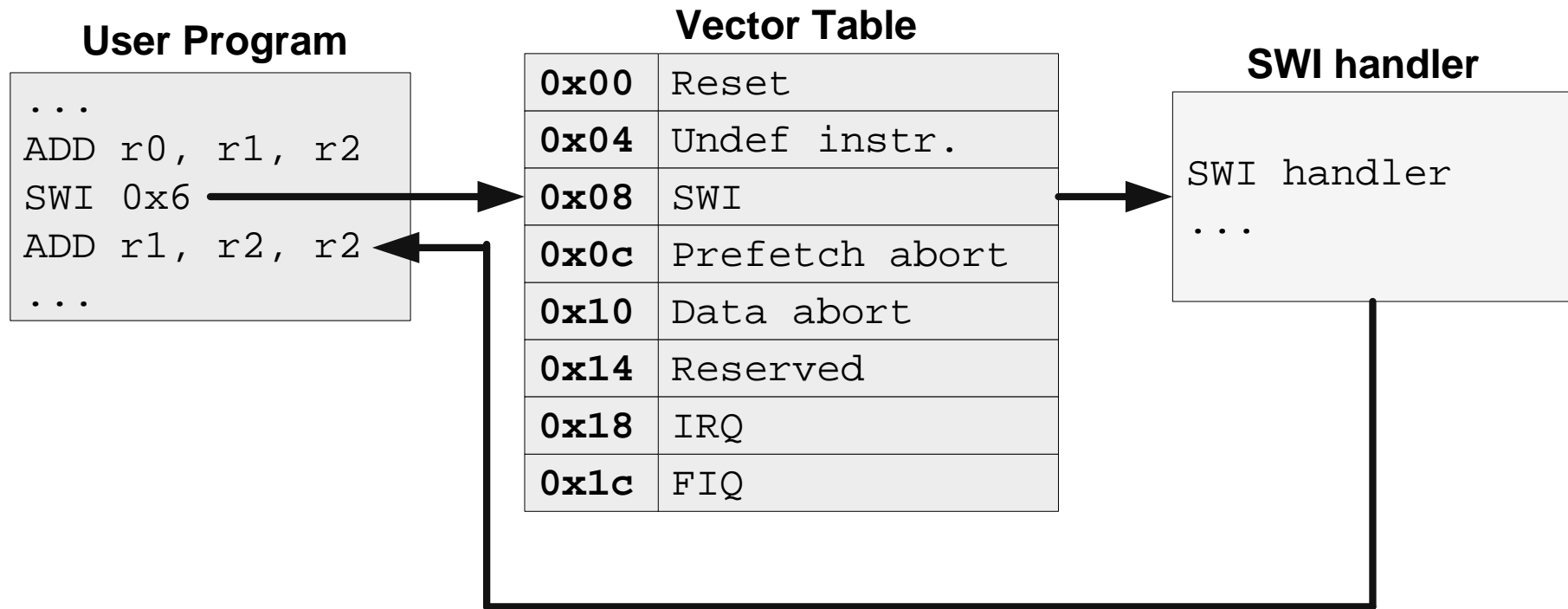
```
; This routine sends the character in the bottom  
; byte of r0 to the use display device
```

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

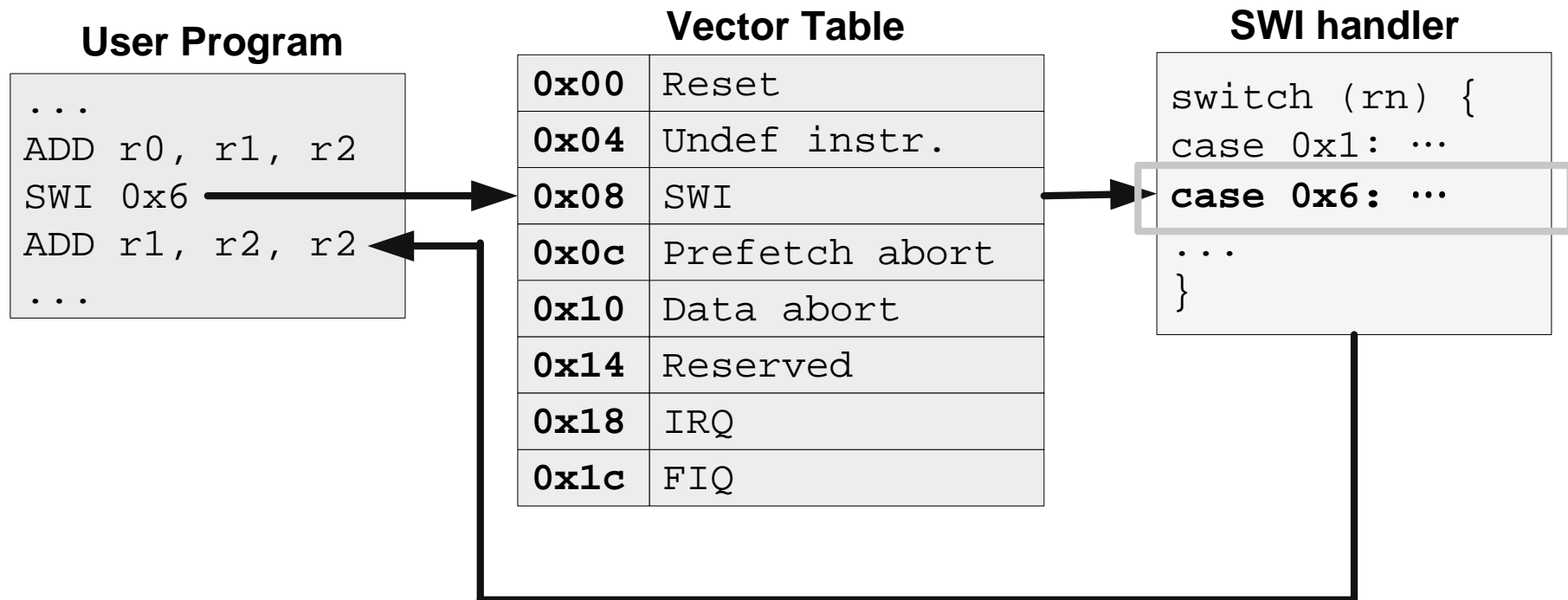
Processor Actions for SWI (1)

- 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)



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
SWI_WriteC    EQU    &0
SWI_Exit      EQU    &11

        ENTRY
START    ADR      r1, TEXT
LOOP     LDRB     r0, [r1], #1
         CMP      r0, #0
         SWINE    SWI_WriteC
         BNE      LOOP
         SWI      SWI_Exit
TEXT     =        "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 (=)

ENTRY: The first instruction to be executed within an application is marked by the ENTRY directive. 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 object 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
```

- LABEL透過":"來做識別
- armasm則是透過指令和保留字的縮排來做識別

Filename: test.s

- 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

DSP Extensions

- DSP Extensions “E”
 - 16bit Multiply and Multiply-Accumulate instructions
 - Saturated, signed arithmetic
 - Introduced in v5TE
 - Available in ARM9E, ARM10E and Jaguar families

ARM Java Extensions - Jazelle™

- Direct execution of Java ByteCode
- 8x Performance of Software JVM
(Embedded CaffeineMark3.0)
- Over 80% power reduction for Java Applications
- Single Processor for Java and existing OS/applications
- Supported by leading Java Run-time environments and operating systems
- Available in ARM9, ARM10 & Jaguar families

ARM Media Extensions (ARM v6)

- Applications
 - Audio processing
 - MPEG4 encode/decode
 - Speech Recognition
 - Handwriting Recognition
 - Viterbi Processing
 - FFT Processing
- Includes
 - 8 & 16-bit SIMD operations
 - ADD, SUB, MAC, Select
- Up to 4x performance for no extra power
- Introduced in ARM v6 architecture, Available in Jaguar

ARM Architectures

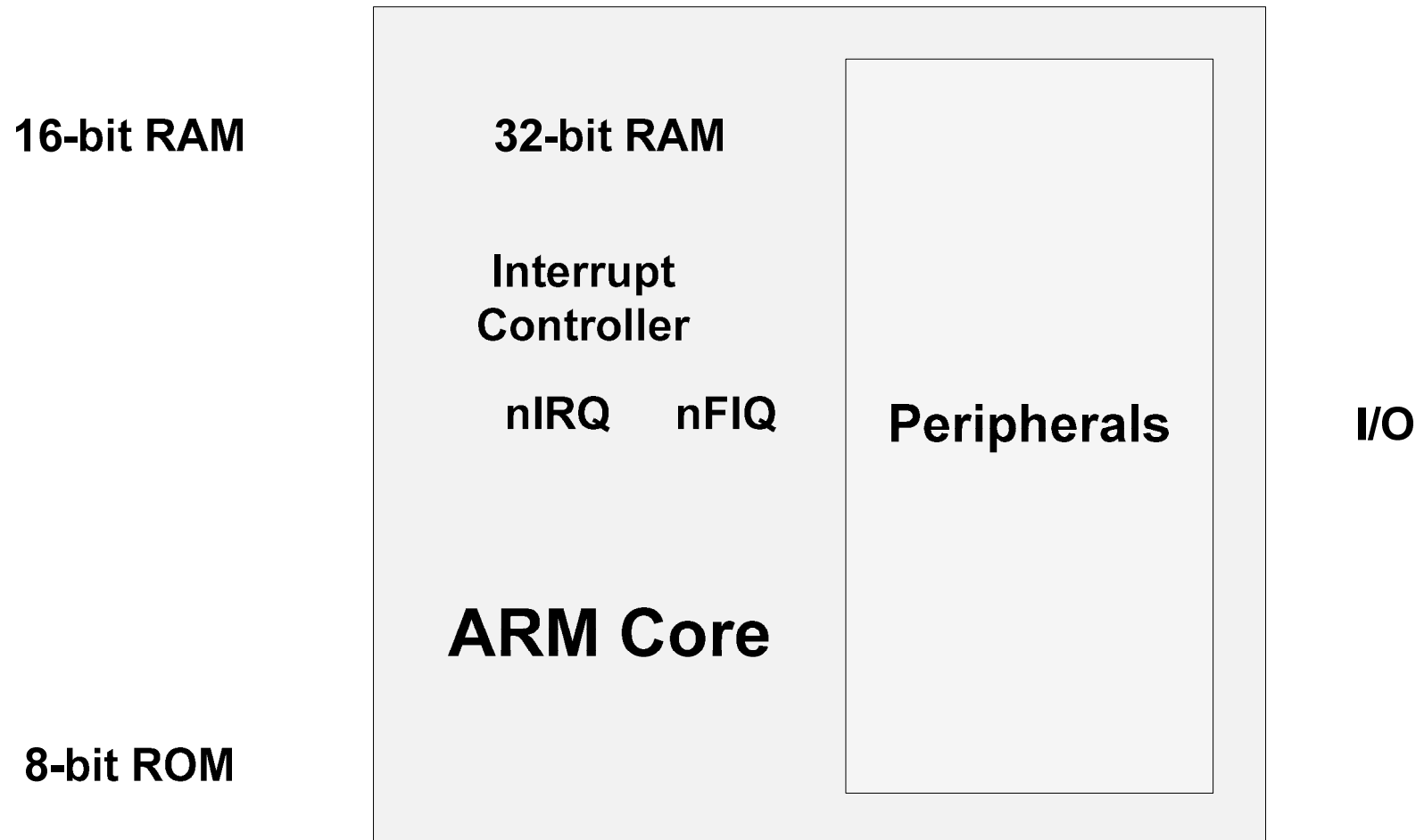
Architecture	Feature Set			
	THUMB™	DSP	Jazelle™	Media
v4T	✓			
v5TE	✓	✓		
v5TEJ	✓	✓	✓	
v6	✓	✓	✓	✓

- Enhance performance through innovation
 - THUMB™: 30% code compression
 - DSP Extensions: Higher performance for fixed-point DSP
 - Jazelle™: up to 8x performance for java
 - Media Extensions up to 4x performance for audio & video
- Preserve Software Investment through compatibility

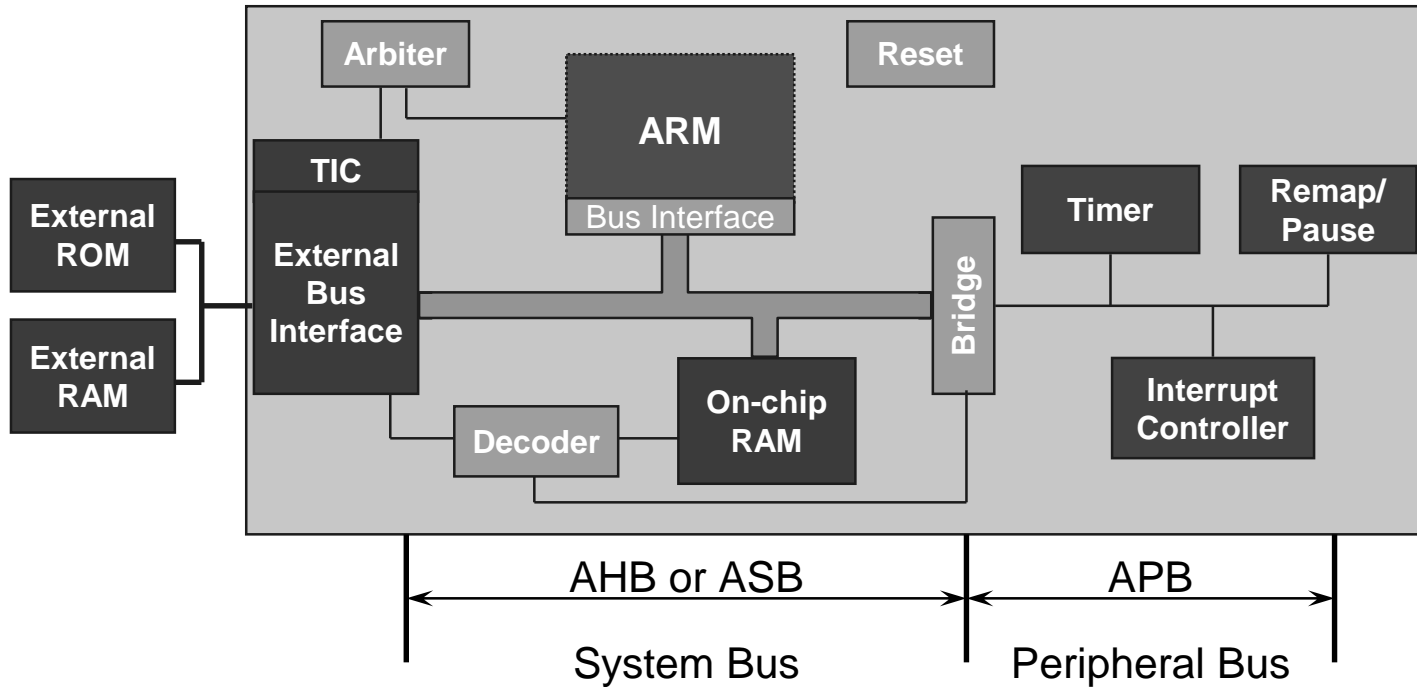
Outline

- Introduction
- Programmers model
- Instruction set
- **System design**
- Development tools

Example ARM-based System



AMBA



- **AMBA**
 - **Advanced Microcontroller Bus Architecture**
- **ADK**
 - **Complete AMBA Design Kit**
- **ACT**
 - **AMBA Compliance Testbench**
- **PrimeCell**
 - **ARM's AMBA compliant peripherals**

reference: <http://www.intel.com/education/highered/modelcurriculum.htm>

ARM Coprocessor Interface

- ARM supports a general-purpose extension of its instructions set through **the addition of hardware coprocessor**
- **Coprocessor architecture**
 - Up to 16 logical coprocessors
 - Each coprocessor can have up to 16 private registers (any reasonable size)
 - Using load-store architecture and some instructions to communicate with ARM registers and memory.

ARM7TDMI Coprocessor Interface

- Based on “bus watching” technique
- The coprocessor is attached to a bus where the ARM instruction stream flows into the ARM
- The coprocessor copies the instructions into an internal pipeline
- **A “hand-shake” between the ARM and the coprocessor confirms that they are both ready to execute coprocessor instructions**

Outline

- Introduction
- Programmers model
- Instruction set
- System design
- **Development tools**

Development Tools (1)

- **Commercial**

- ARM

- IAR

- ...

← **Best code quality**

- **Open source**

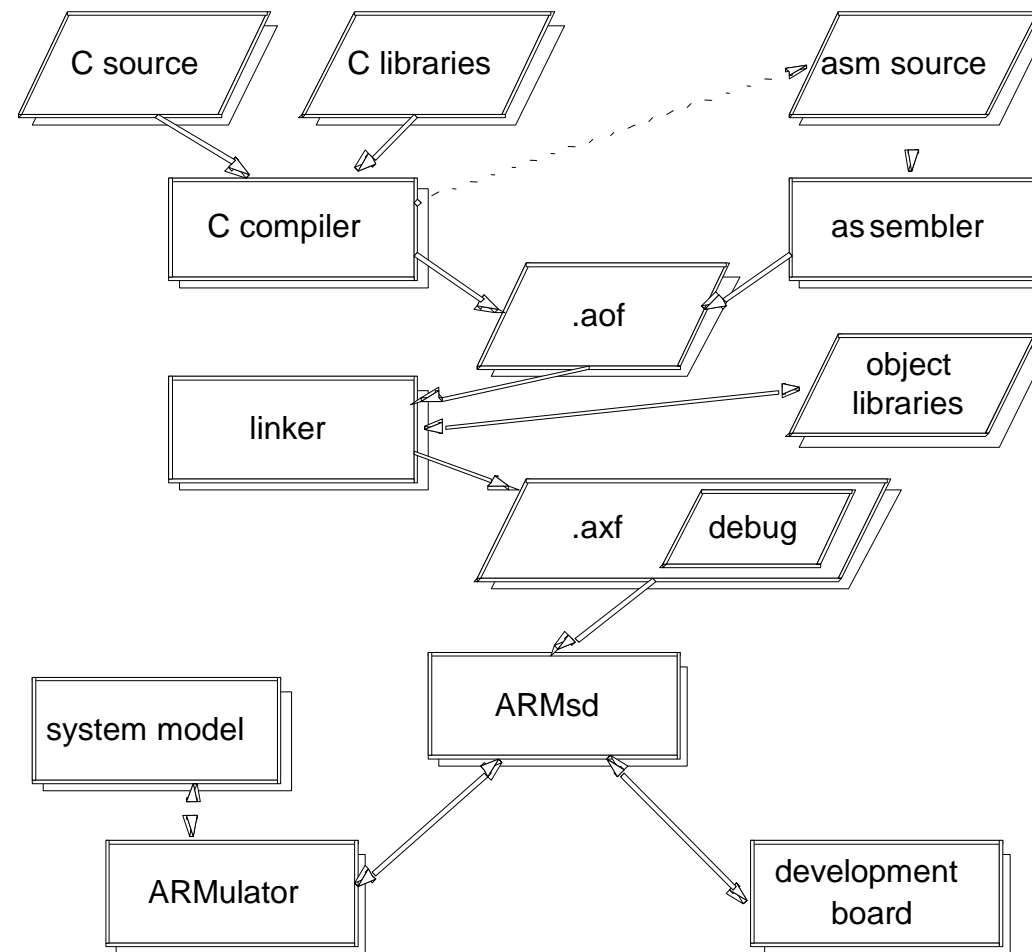
- GNU



Development Tools (2)

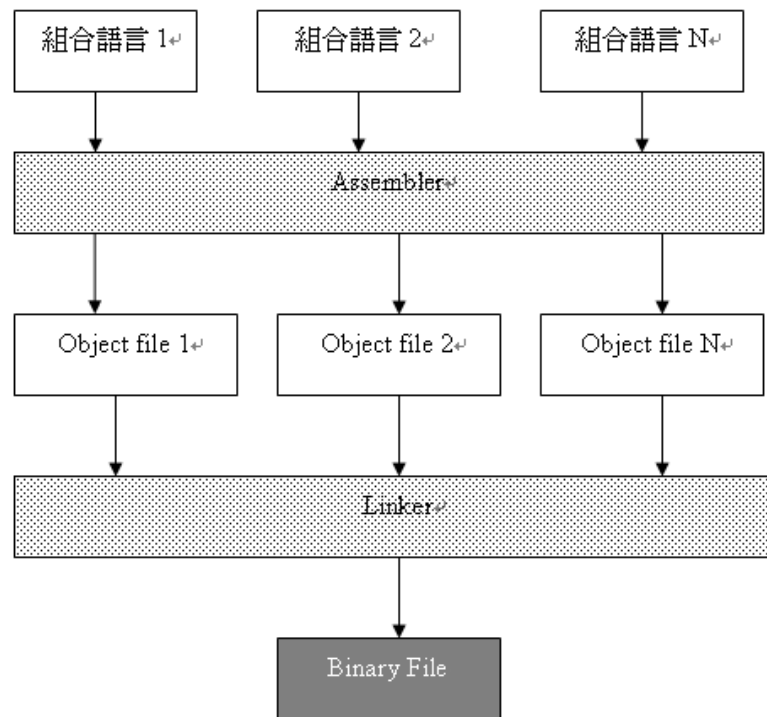
	ARM ADS	GNU
Compiler	armcc	gcc
Assembler	armasm	binutils
Linker	armlink	binutils
Format converter	fromelf	binutils
C library	C library	newlib
Debugger	Armsd, AXD	GDB, Insight
Simulator	ARMulator	Simulator in GDB

The Structure of ARM Cross-Development Toolkit



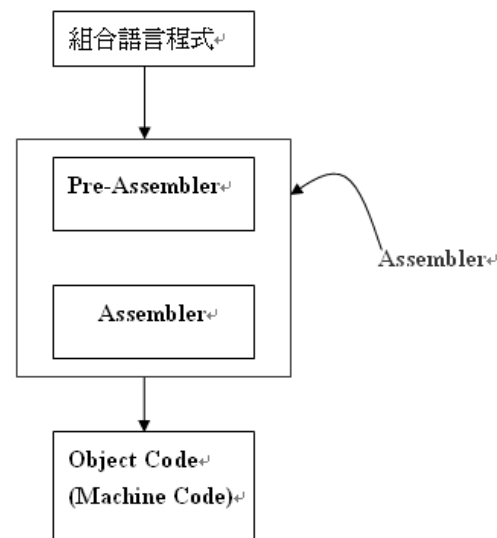
ADS-Assembler

- Compiler：產生Object
- Linker：產生**ELF** 可執行碼



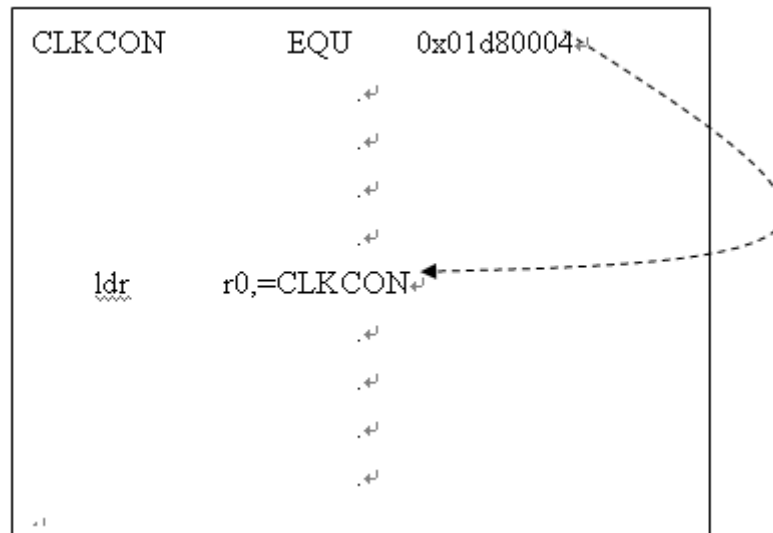
ADS- Pre-assembler

- Pre-assembler
 - Pseudo code -> assembler -> Object



Example

- Example of pr-compiler



Example

- Example of pr-compiler

