

# **The ARM Assembly Language Programming**

**Peng-Sheng Chen**

Fall, 2017

# Introduction

- The ARM processor is **easy** to program at the assembly level
- In this study, we will
  - **Look at ARM assembly language programming at the user level**
  - **See how to write simple programs which will run on an ARM development board / ARM emulator**

# Outline

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

# Outline

- **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

# 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] OPlogic 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



# Example: BIC Instruction

Assume that r1, r2, and r0 are 16-bit registers

- $r1 = 1111111111111111$   
 $r2 = 0000000001100101$

**BIC r0, r1, r2**

- $r0 = 1111111110011010$

# 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

# 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, #0xff   ; r8 := r7[7:0]
```

A constant preceded by '#'

A hexadecimal by putting '0x' 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
-----	--------------------------------	---



**LSR #5**

# 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)
-----	-----------------------------------	---

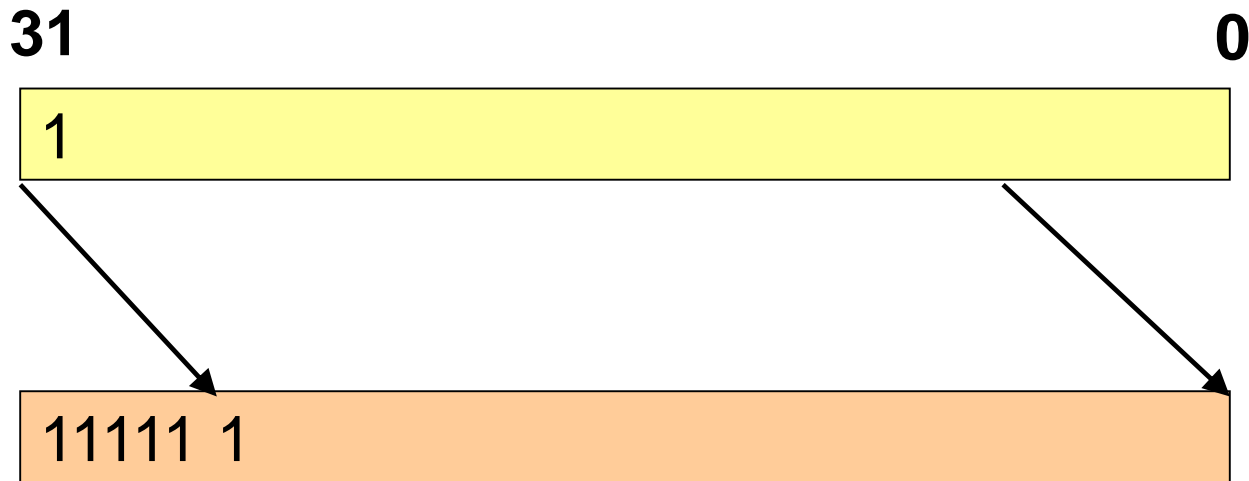


**ASR #5 ;positive operand**



# 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**

# Unsigned Integer

- 16 bit

– 0000000000000000  $\Rightarrow 0$

– 1111111111111111  $\Rightarrow 2^{16} - 1 = 65535$

# Signed Integer (2's Complement)

- 32767  $\Rightarrow$  **0**111111111111111111 (最大正數)
- 4  $\Rightarrow$  **0**0000000000000000100
- 1  $\Rightarrow$  **0**0000000000000000001
- 0  $\Rightarrow$  **0**0000000000000000000
- -1  $\Rightarrow$  **1**1111111111111111111
- -4  $\Rightarrow$  **1**1111111111111111000
- -32767  $\Rightarrow$  **1**0000000000000000001
- -32768  $\Rightarrow$  **1**0000000000000000000 (最小負數)

# Example 1

- 8  $\Rightarrow$  0000000000000001000
- Shift right 2 bits
  - 0000000000000001000
  - 000000000000000010 (LSR)
  - 000000000000000010 (ASR)

# Example 2

- $-8 \Rightarrow$  **1**11111111111111000
- Shift right 2 bits
  - **1**11111111111111000
  - **00****1**11111111111110 (LSR)
  - **11****1**11111111111110 (ASR)

# Shifted Register Operands (6)

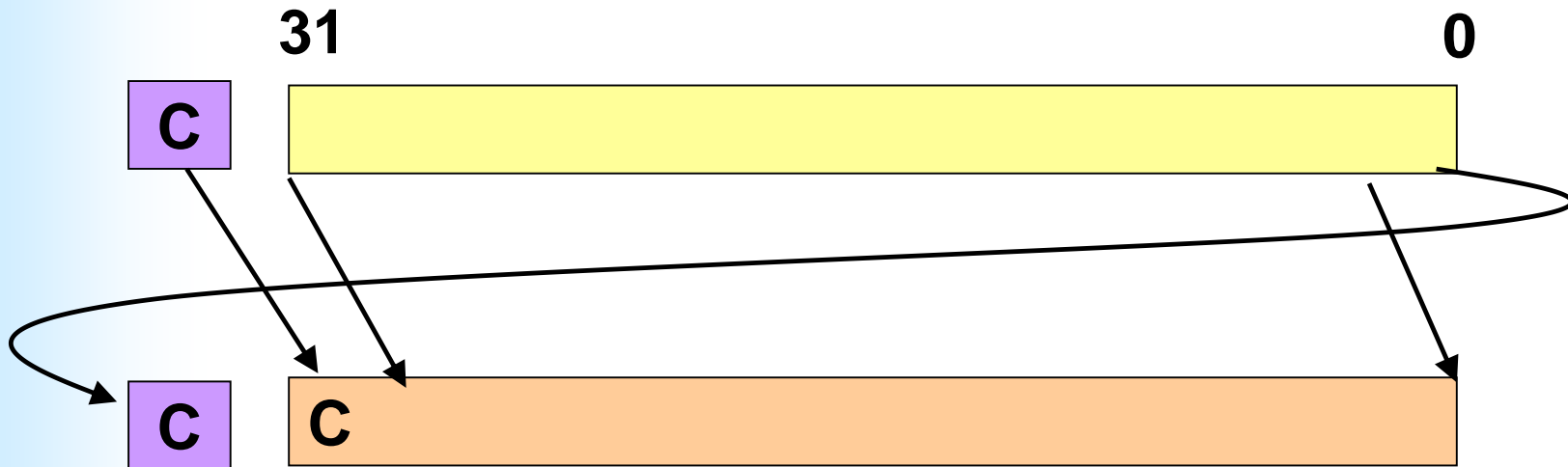
ROR	Rotate right by 0 to 32	The bits which fall off the LSB of the word are used to fill the vacated bits at the MSB of the word
-----	-------------------------	--



**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:

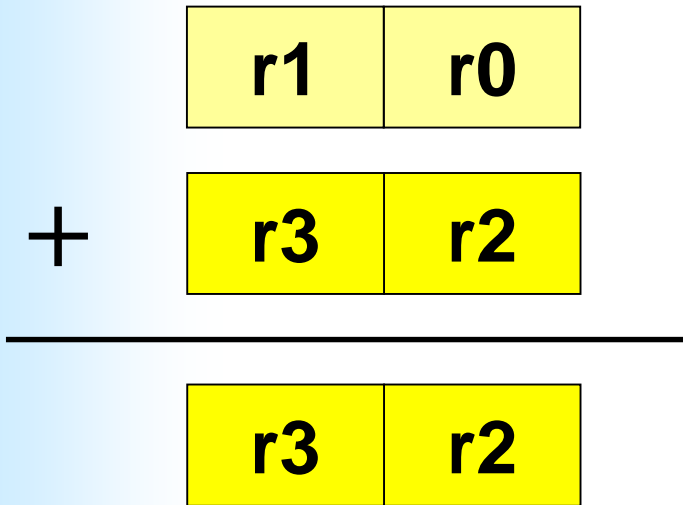
```
ADD    r5, r5, r3, LSL r2    ; r5:=r5+r3*2^r2
```

- 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 3

```
; move 3 to r1  
MUL    r3, r0, r1 ; r3 := r0 x 3
```

OR

```
ADD    r3, r0, r0, LSL #1 ; r3 := r0 + r0 x 2
```

# Multiplies (3)

- Ex: multiply r0 by 2

```
; move 2 to r1  
MUL    r3, r0, r1 ; r3 := r0 x 2
```

OR

```
MOV     r3, r0, LSL #1 ; r3 := r0 x 2
```

- 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'
```

# Outline

- 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 and Store Instructions (1)

- These instructions provide the most flexible way to transfer single data item 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
```

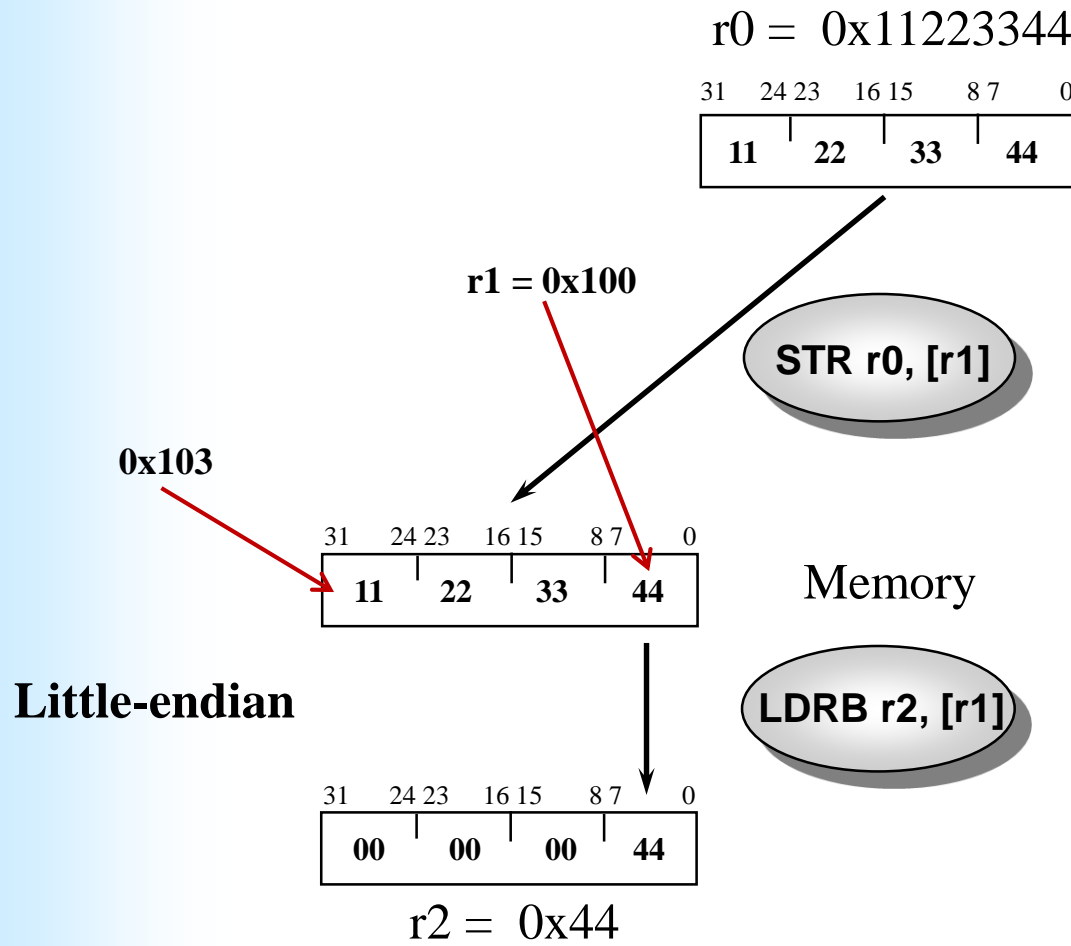
**Register-indirect addressing**



# Single Register Load and Store Instructions (2)

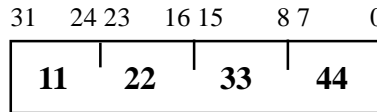
<b>LDR</b>	Load a word into register	$Rd \leftarrow \text{mem32}[\text{address}]$
<b>STR</b>	Store a word in register into memory	$\text{Mem32}[\text{address}] \leftarrow Rd$
<b>LDRB</b>	Load a byte into register	$Rd \leftarrow \text{mem8}[\text{address}]$
<b>STRB</b>	Store a byte in register into memory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

# Endianness Example



# Endianness Example

$r0 = 0x11223344$

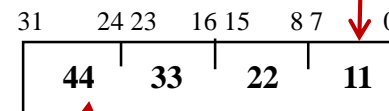


**STR  $r0$ ,  $[r1]$**

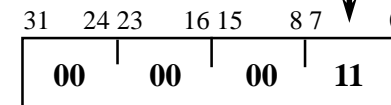
Memory

**LDRB  $r2$ ,  $[r1]$**

$r1 = 0x100$



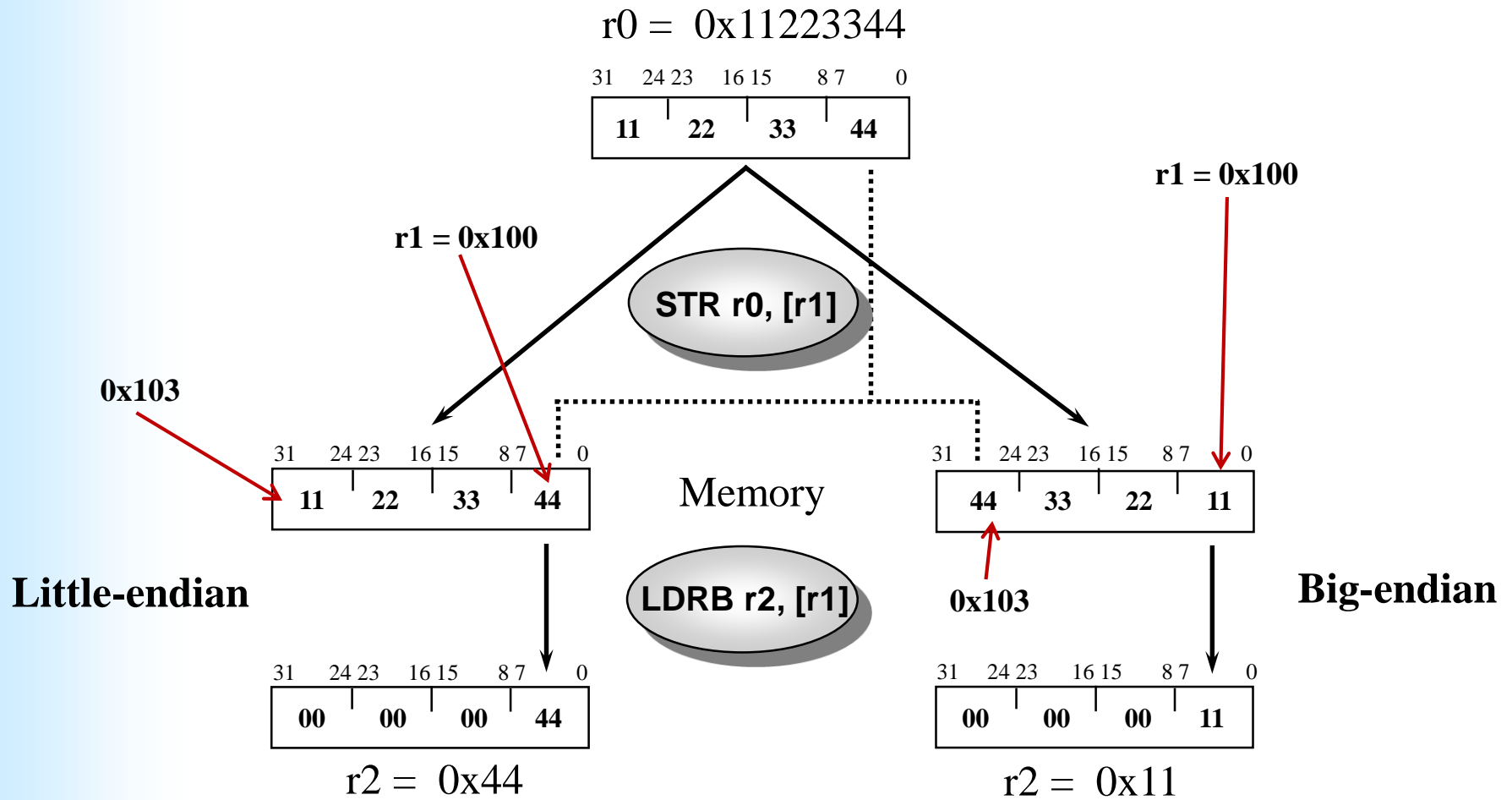
$0x103$



$r2 = 0x11$

**Big-endian**

# Endianness Example



# 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
```

```

i = 0;
while (i < n) {
    do some operation on A[i];
    i ++;
}

```

# Application

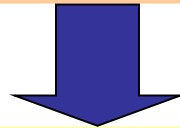
table

Memory
0x100
A[0]
A[1]
A[2]

```

ADR    r1, table
LOOP   LDR    r0, [r1]          ; r0 := mem32[r1]
      ADD    r1, r1, #4        ; r1 := r1 + 4
      ;
      ;do some operation on A[i] (that is r0)
      ...

```



```

ADR    r1, table
LOOP   LDR    r0, [r1], #4      ; r0 := mem32[r1]
      ; r1 := r1 + 4
      ;
      ;do some operation on A[i] (that is r0)
      ...

```



# Example

- **Pre-indexed addressing mode**

```
LDR    r0, [r1, #8]    ; r0 := mem32[r1 + 8]
```

- **Auto-indexing (Preindex with writeback)**

```
LDR    r0, [r1, #8]!   ; r0 := mem32[r1 + 8]  
                        ; r1 := r1 + 8
```

- **Post-indexed addressing mode**

```
LDR    r0, [r1], #8    ; r0 := mem32[r1]  
                        ; r1 := r1 + 8
```

# Multiple Register Load and 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 and Store Instructions (2)

<b>LDM</b>	Load multiple registers
<b>STM</b>	Store multiple registers

Addressing mode	Description	Starting address	End address	Rn!
<b>IA (increase after)</b>	執行後增加	$R_n$	$R_n + 4 * N - 4$	$R_n + 4 * N$
<b>IB (increase before)</b>	執行前增加	$R_n + 4$	$R_n + 4 * N$	$R_n + 4 * N$
<b>DA (decrease after)</b>	執行後減少	$R_n - 4 * R_n + 4$	$R_n$	$R_n - 4 * N$
<b>DB (decrease before)</b>	執行前減少	$R_n - 4 * N$	$R_n - 4$	$R_n - 4 * N$

Addressing mode for multiple register load and store instructions

# Example (1)

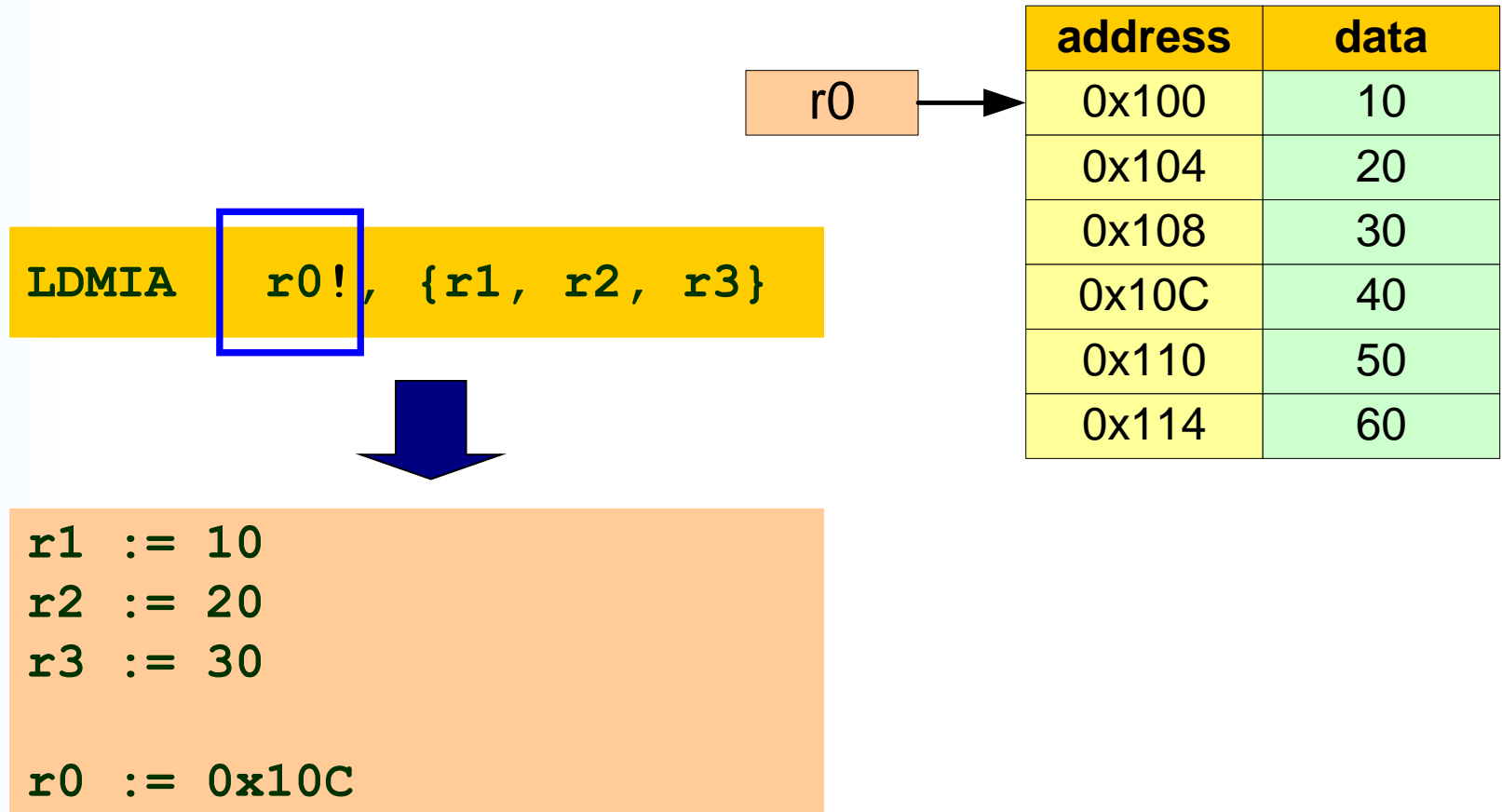
```
LDMIA    r0, {r1, r2, r3}  
OR  
LDMIA    r0, {r1-r3}
```

r0 →

address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x110	50
0x114	60

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

# Example (2)

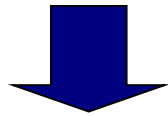


# Example (3)

r0 →

address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x110	50
0x114	60

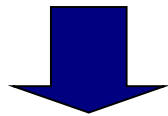
```
LDMIB  r0!, {r1, r2, r3}
```



```
r1 := 20  
r2 := 30  
r3 := 40  
  
r0 := 0x10C
```

# Example (4)

**LDMDA**    **r0!, {r1, r2, r3}**



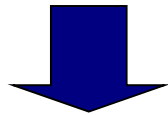
r0 →

address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x110	50
0x114	60

**r1 := 40**  
**r2 := 50**  
**r3 := 60**  
  
**r0 := 0x108**

# Example (5)

**LDMDB**    **r0!, {r1, r2, r3}**



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

r0 := 0x108
```

r0 →

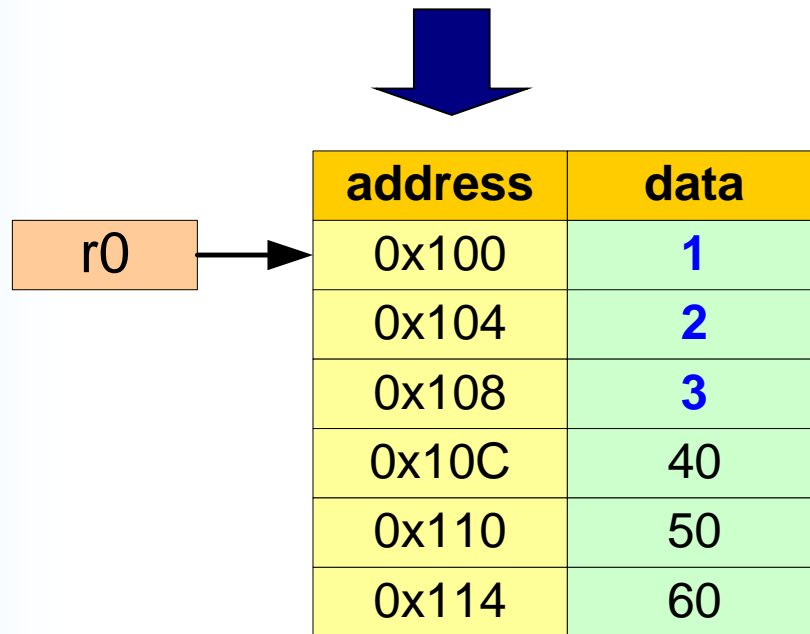
address	data
0x100	10
0x104	20
0x108	30
0x10C	40
0x110	50
0x114	60



# Example (6)

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

```
r1 := 1  
r2 := 2  
r3 := 3  
  
r0 := 0x100
```



# Example (7)

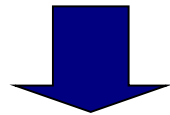
STMIA **r0!**, {r1, r2, r3}  
OR  
STMIA **r0!**, {r1-r3}

r1 := 1

r2 := 2

r3 := 3

r0 := 0x100



r0

address	data
0x100	1
0x104	2
0x108	3
0x10C	40
0x110	50
0x114	60

# Example (8)

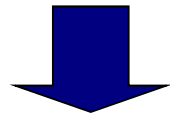
STMIB **r0!**, {r1, r2, r3}  
OR  
STMIB **r0!**, {r1-r3}

r1 := 1

r2 := 2

r3 := 3

r0 := 0x100



r0

address	data
0x100	10
0x104	1
0x108	2
0x10C	3
0x110	50
0x114	60

# Example (9)

STMDA **r0!**, {r1, r2, r3}  
OR  
STMDA **r0!**, {r1-r3}

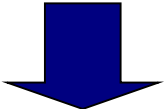
r1 := 1

r2 := 2

r3 := 3

r0 := 0x110

r0



address	data
0x100	10
0x104	20
0x108	<b>1</b>
0x10C	<b>2</b>
0x110	<b>3</b>
0x114	60

# Example (10)

STMDB **r0!**, {r1, r2, r3}  
OR  
STMDB **r0!**, {r1-r3}

r1 := 1

r2 := 2

r3 := 3

r0 := 0x110

r0 →

address	data
0x100	10
0x104	<b>1</b>
0x108	<b>2</b>
0x10C	<b>3</b>
0x110	50
0x114	60

# Multiple Register Load and Store Instructions (3)

- Base register used to determine where memory access should occur
  - 4 different addressing modes allow increment and decrement inclusive or exclusive of the base register location.
  - Base register can be optionally updated following the transfer by appending it with an '!'
  - Lowest register number is always transferred to/from lowest memory location accessed

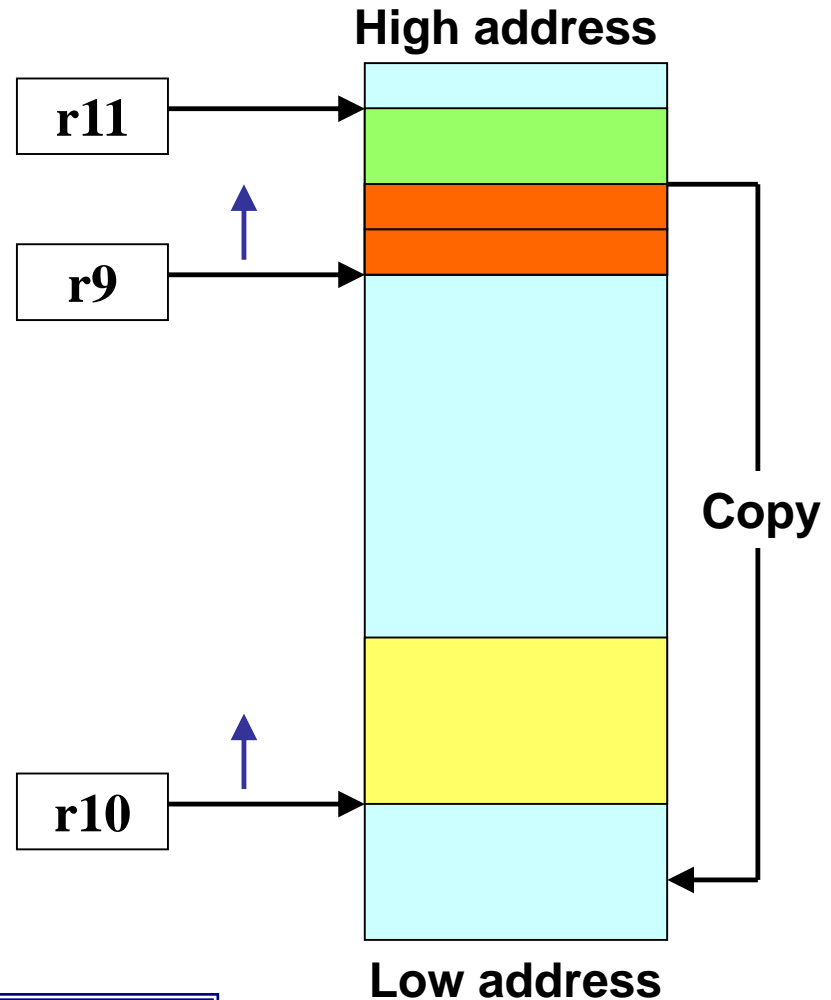
# Application

Copy a block of memory

```
; r9  存放來源資料的起始位址  
; r10 存放目標的起始位址  
; r11 存放來源資料的結束位址
```

LOOP

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



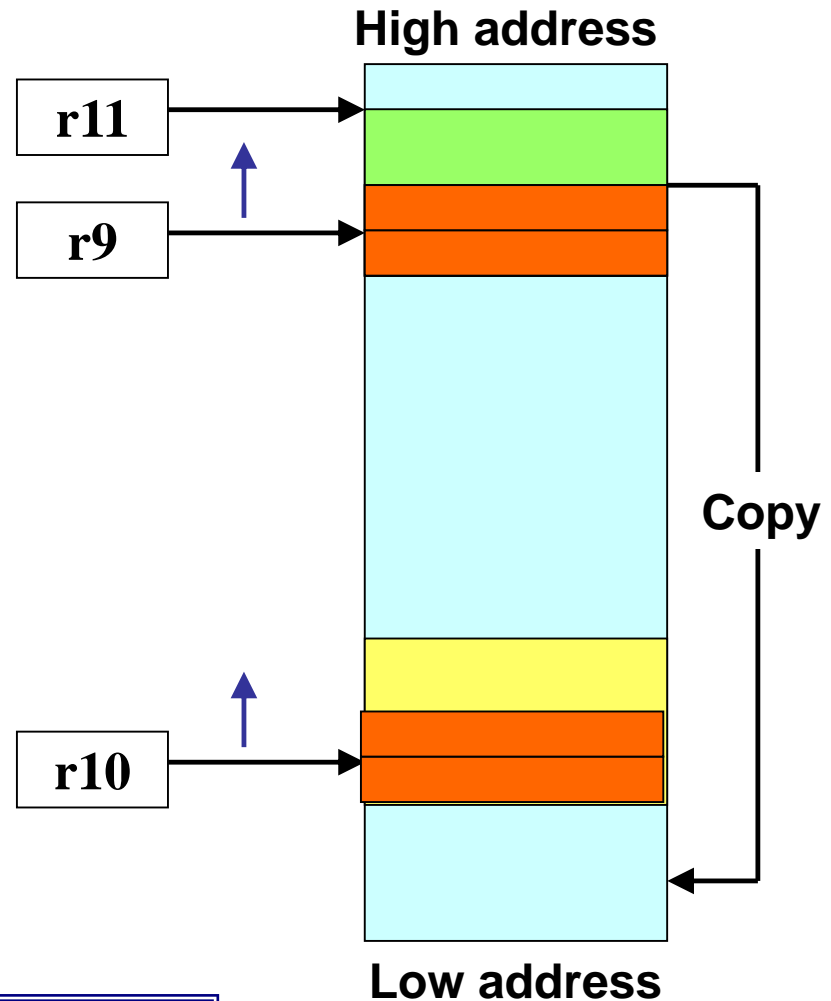
# Application

Copy a block of memory

```
; r9  存放來源資料的起始位址  
; r10 存放目標的起始位址  
; r11 存放來源資料的結束位址
```

LOOP

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





# Application: Stack Operations

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

# The Stack (1)

- Stack向上生長或向下生長
  - Ascending, 'A': 遞增
  - Descending, 'D': 遞減
- Full stack, 'F': sp指向stack的最後一個已使用的位址
- Empty stack, 'E': sp指向stack的第一個沒有使用的位址

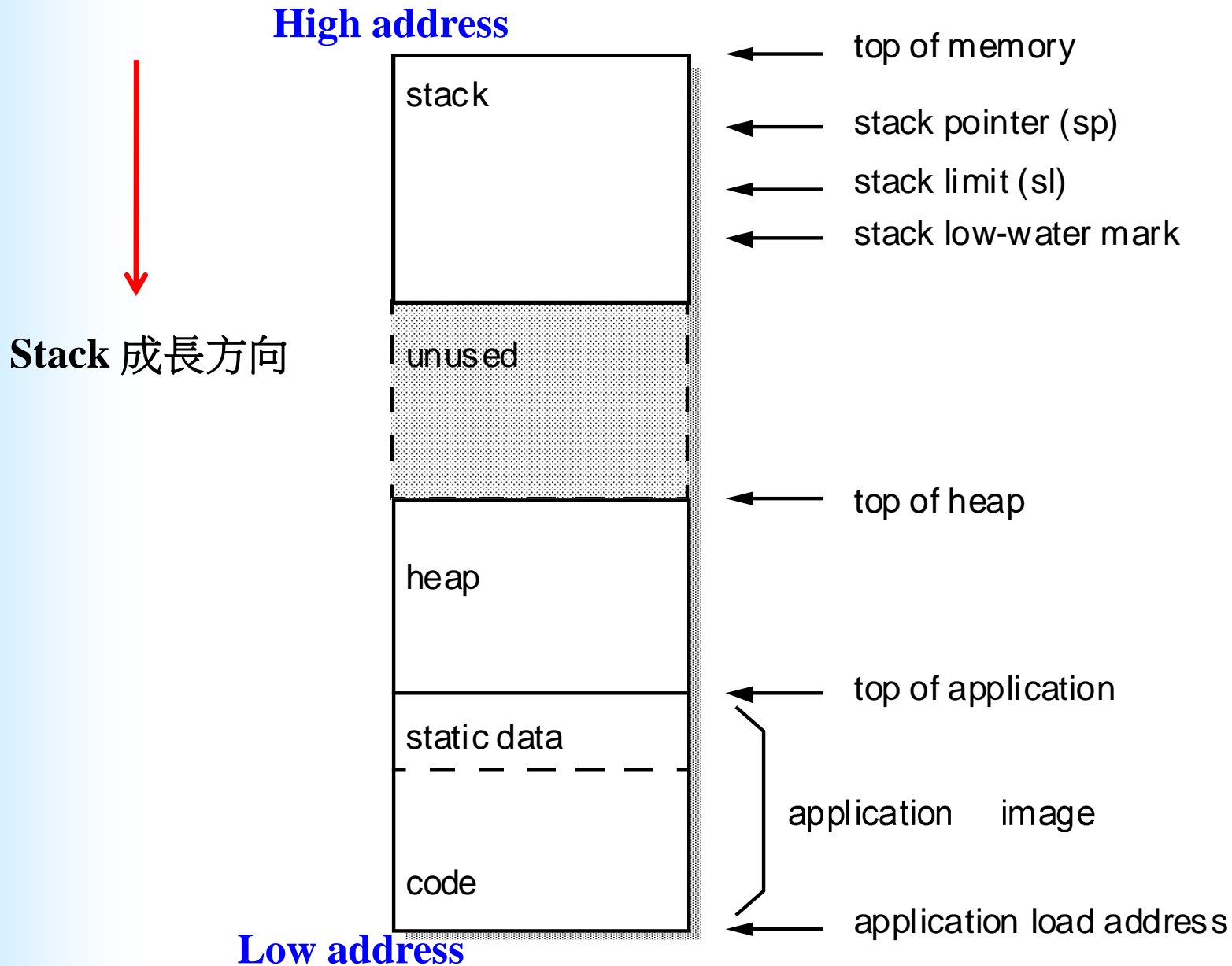
# The Stack (2)

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

定址方式	說明	POP	=LDM	PUSH	=STM
FA	遞增滿	LDMFA	LDMDA	STMFA	STMIB
FD	遞減滿	LDMFD	LDMIA	STMFD	STMDB
EA	遞增空	LDMEA	LDMDB	STMEA	STMIA
ED	遞減空	LDMED	LDMIB	STMED	STMDA

# The Stack (3)

- The stack type to be used is given by the **postfix** to the instruction:
  - **STMFD/LDMFD: Full Descending stack**
  - STMFA/LDMFA: Full Ascending stack
  - STMED/LDMED: Empty Descending stack
  - STMEA/LDMEA: Empty Ascending stack
- Pseudo instruction
- Note: ARM Compilers will always use a **Full descending stack**



定址方式	說明	POP	=LDM	PUSH	=STM
FA	遞增滿	LDMFA	LDMDA	STMFA	STMIB

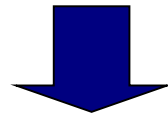
STMFA **sp!**, {r1, r2, r3}  
OR  
STMFA **sp!**, {r1-r3}

```

r1 := 1
r2 := 2
r3 := 3

sp := 0x100

```



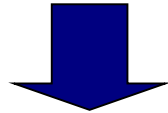
	address	data
	0x100	10
	0x104	<b>1</b>
	0x108	<b>2</b>
sp →	0x10C	<b>3</b>
	0x110	50
	0x114	60

定址方式	說明	POP	=LDM	PUSH	=STM
FA	遞增滿	LDMFA	LDMDA	STMFA	STMIB

LDMFA  
OR  
LDMFA

**sp!**, {r4, r5, r6}  
**sp!**, {r4-r6}

sp := 0x10C



sp →

address	data
0x100	10
0x104	<b>1</b>
0x108	<b>2</b>
0x10C	<b>3</b>
0x110	50
0x114	60

r4 := 1  
r5 := 2  
r6 := 3

sp := 0x100

# 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
  - 在操作期間，禁止其他指令對欲存取的儲存單元讀寫
- Application
  - Implement semaphores (multi-threaded / multi-processor environment)

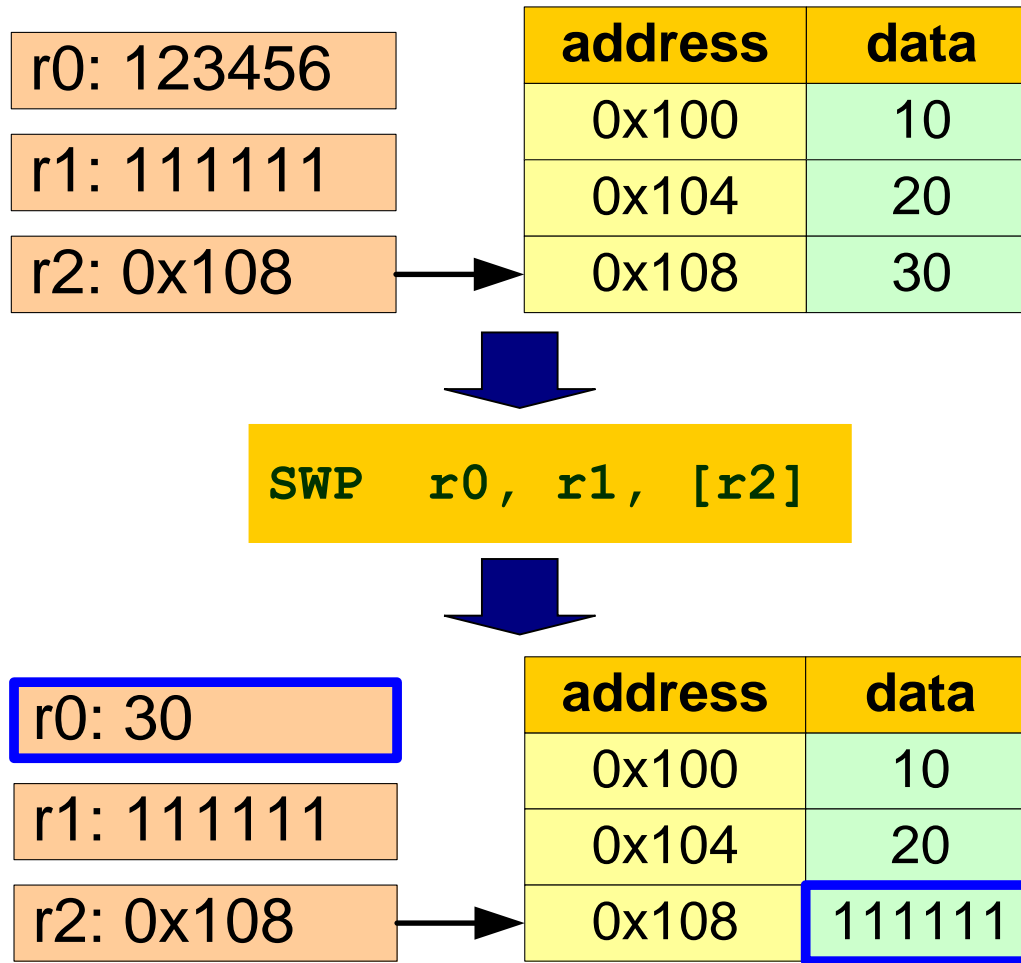


# Single Register Swap Instructions (2)

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

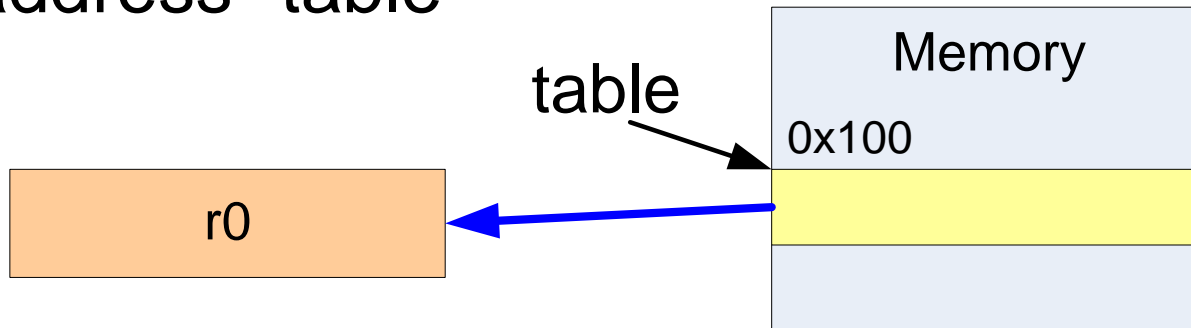
SWP	WORD exchange	tmp = mem32[Rn] mem32[Rn] = Rm Rd = tmp
SWPB	Byte exchange	tmp = mem8[Rn] mem8[Rn] = Rm Rd = tmp

# Example



# 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	ldrvcht	r6,[r3],#-0x574
0x000080b0	andeq	r0,r0,r2,lsr r0
xyz	ldrvcht	r6,[r3],#-0x574
_edata	andeq	r0,r0,r0

Program terminated normally

ARMulate

# Outline

- 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  ...
      ;do something here
      ...
      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	Conditional execution
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 & 0xffffffe, T=Rm & 1
BLX	帶返回的跳躍並 切換狀態	PC=label, T=1 PC=Rm & 0xffffffe, 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
```

# Example

r14: (lr) => 0x104

0x100  
0x104

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

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

# Example

0x100

0x104

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

subroutine

...

```
MOV     pc, lr      ; return
```

r15 (pc) => 0x104

# Example

0x100

0x104

```
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

# Problem

**0x80**      BL      SUB1    ; branch to subroutine SUB1  
             SUB      r1, r2, #100



SUB1  
         MOV          r0, r1  
         BL            SUB2  
**0x104**      ADD          r1, r2, r3  
         MOV          pc, r14    ; copy r14 into r15 to return

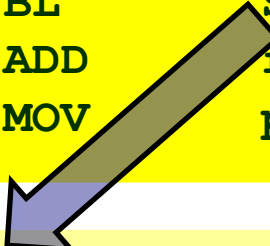
**r14 = 0x80**

SUB2  
         ...  
         MOV          pc, r14    ; copy r14 into r15 to return

# Problem

**0x80**      BL      SUB1    ; branch to subroutine SUB1  
             SUB      r1, r2, #100

**0x104**      SUB1  
             MOV          r0, r1  
             BL          SUB2  
             ADD          r1, r2, r3  
             MOV          pc, r14    ; copy r14 into r15 to return



SUB2  
...  
MOV      pc, r14    ; copy r14 into r15 to return

**r14 = 0x104**



# Branch and Link Instructions (2)

- **Solution**

- Push **r14** into a stack
- The subroutine will often 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

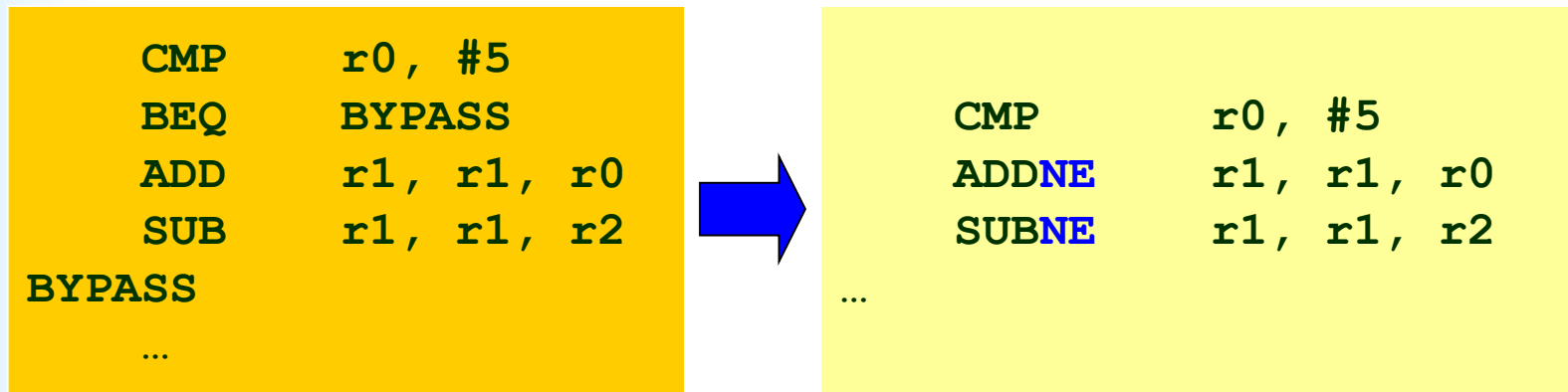
```
STMFD   r13!, {r0-r2, r14} ; save work & link register
BL      SUB2
...
LDMFD   r13!, {r0-r2, pc}  ; restore work register and
                           ; return
```

SUB2

```
...
MOV     pc, r14 ; copy r14 into r15 to return
```

# 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  
      BNE      LABEL1  
      CMP      r2, r3  
      BNE      LABEL1  
      ADD      r4, r4, #1
```

```
LABEL1:
```

# 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
```

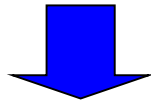
```
CMP      r0, r1  
CMPEQ    r2, r3  
ADDEQ    r4, r4, #1
```

# Conditional Execution (3)

- Predicate
- Real products
  - Partial prediction support
    - SPARC, Alpha, ELF
  - Full prediction support
    - IA-64, XScale, TIC6, ARM

# Conditional Execution (4)

```
if (r1)                // there is a branch b
    add r2, r3, r4;
else
    add r2, r2, 1;
sub r5, r2, r6;         // instruction that behind
                        // branch b
```



```
cmp.ne p0, p1, r1, 0; // branch b: set predicate register
add r2, r3, r4 (p0);   // if p0 is true, r2 = r3 + r4
add r2, r2, 1 (p1);    // if p1 is true, r2 = r2 + 1;
sub r5, r2, r6;
```

## An example of IA64

# Supervisor Calls (1)

- 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]
```



# Supervisor Calls (2)

```
; This routine returns control from a user program  
; back to the monitor program
```

```
SWI      SWI_Exit    ; return to monitor
```

# 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

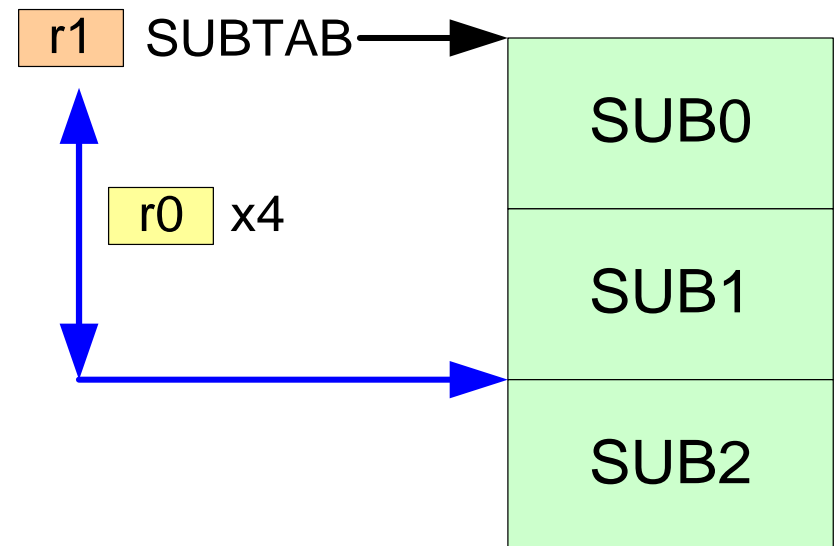
**Note:** slow when the list is long, and all subroutines are equally frequent

```
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
..
```



# Outline

- 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.

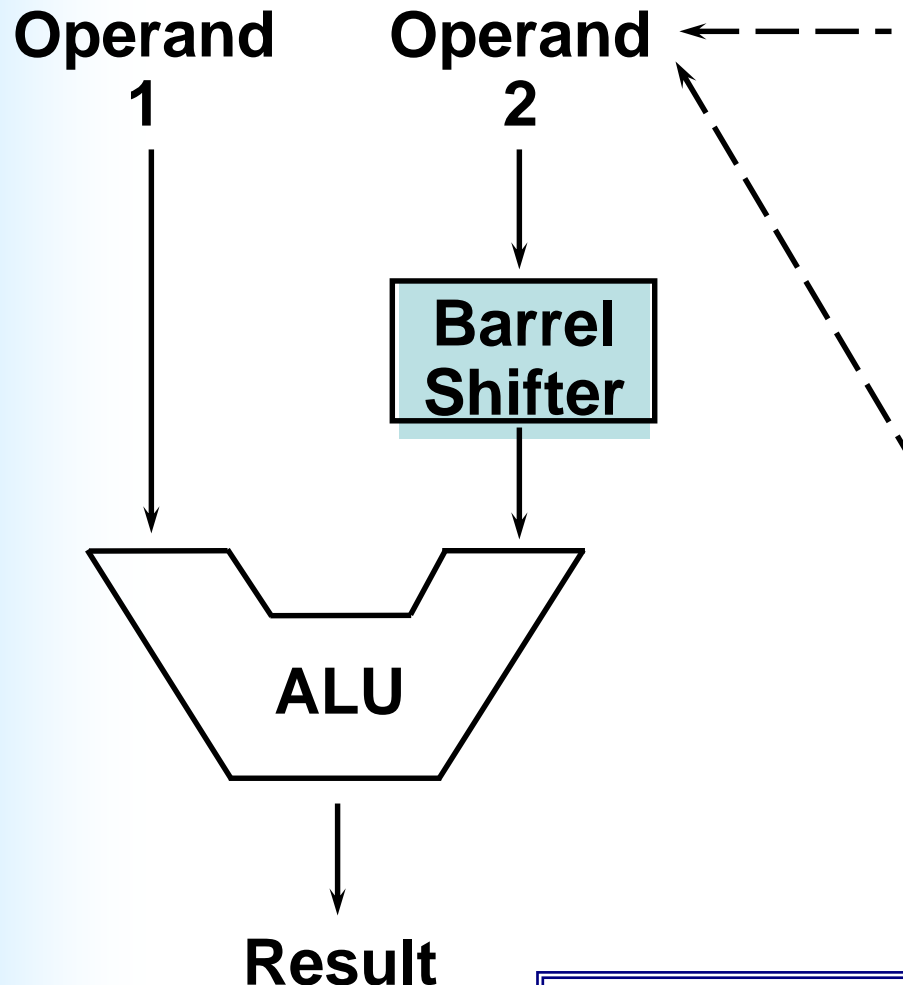
# General Assembly Form (ARM ADS)

```
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

# Backup

# Using the Barrel Shifter: The Second Operand



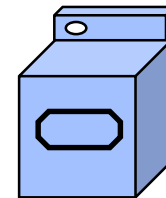
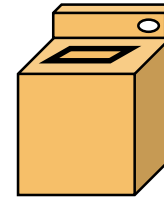
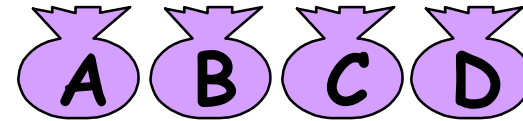
- Register, optionally with shift operation applied.
- Shift value can be either be:
  - 5 bit unsigned integer
  - Specified in bottom byte of another register.

\* **Immediate value**

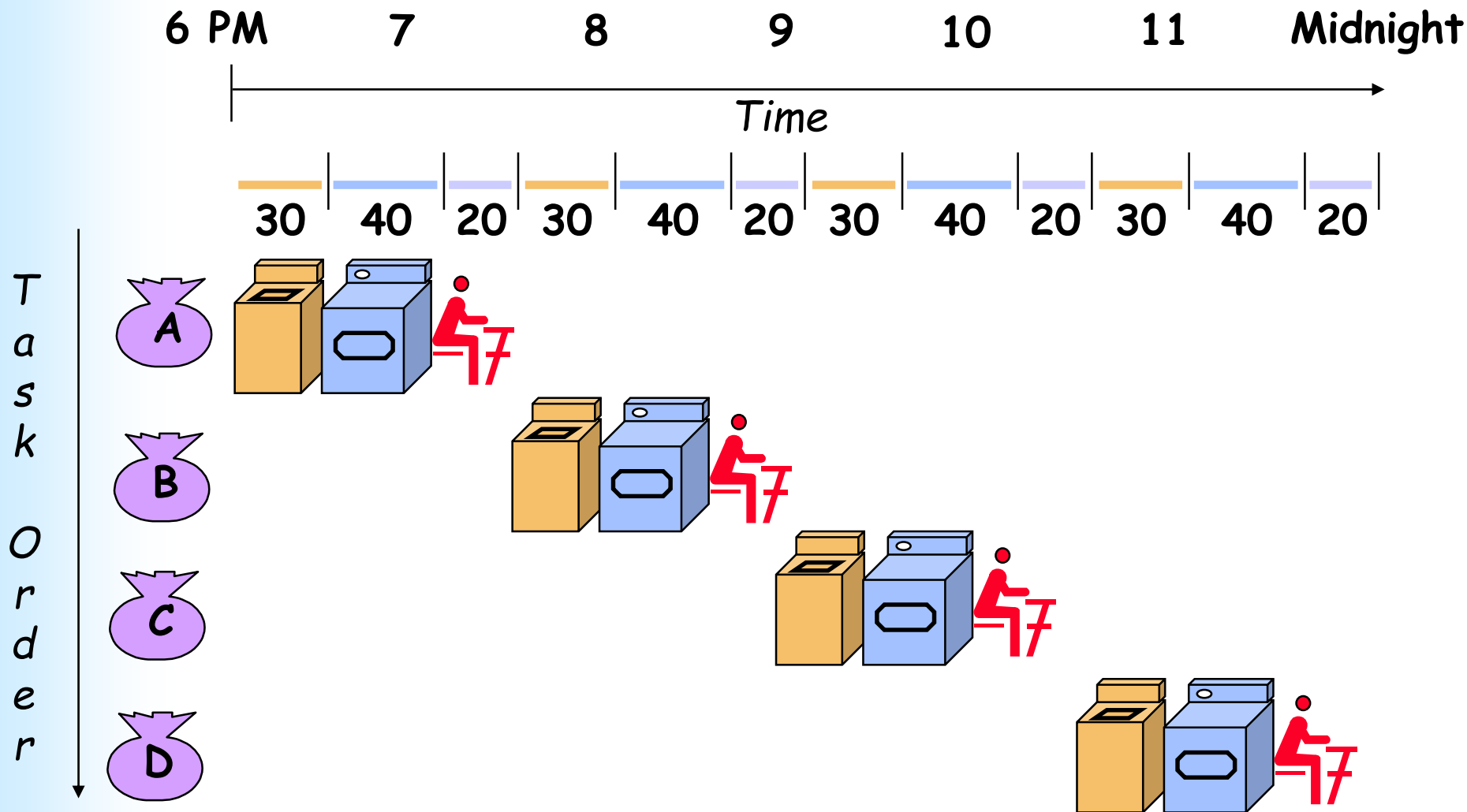


# Example: Pipelines (1)

- Laundry Example
- 4 load of clothes
  - Washer takes 30 minutes
  - Dryer takes 40 minutes
  - “Folder” takes 20 minutes

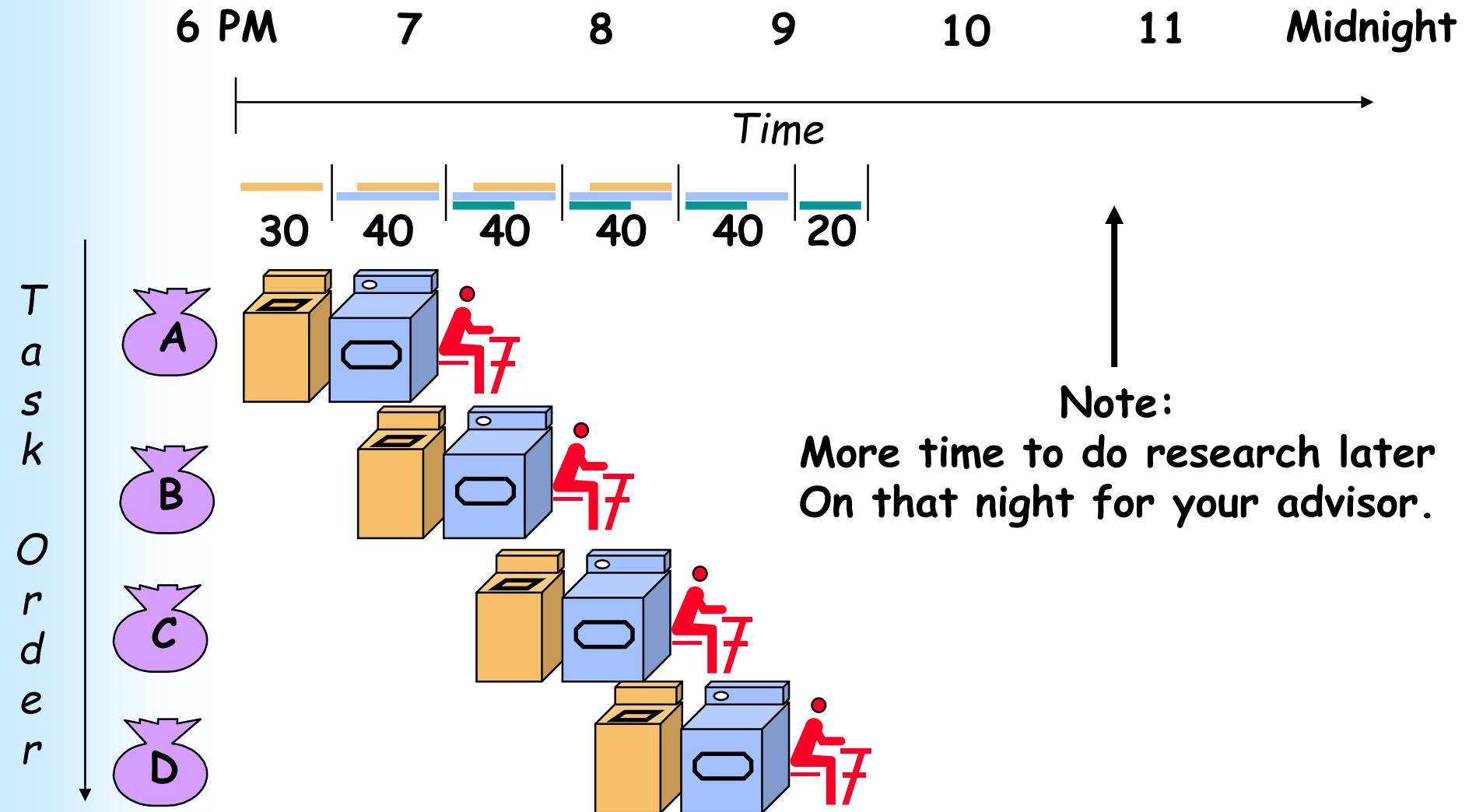


# Sequential Laundry



- Sequential laundry takes 6 hours for 4 loads
- If they learned pipelining, how long would laundry take?

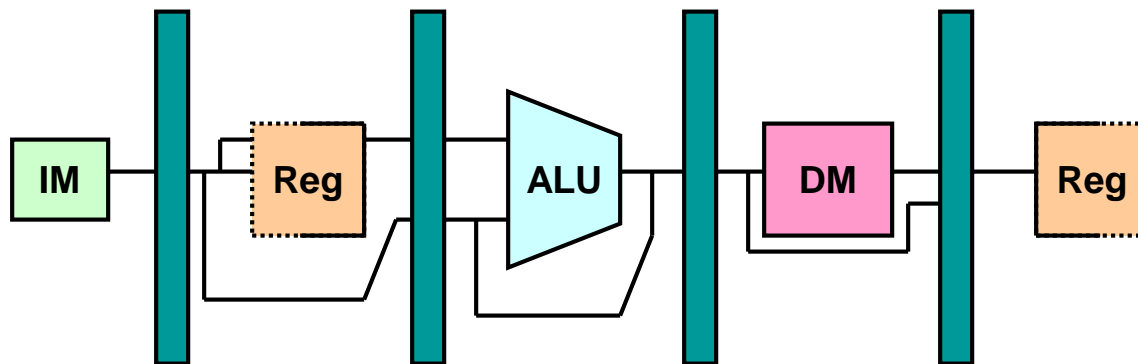
# Pipelined Laundry: Start work ASAP



- Pipelined laundry takes 3.5 hours for 4 loads

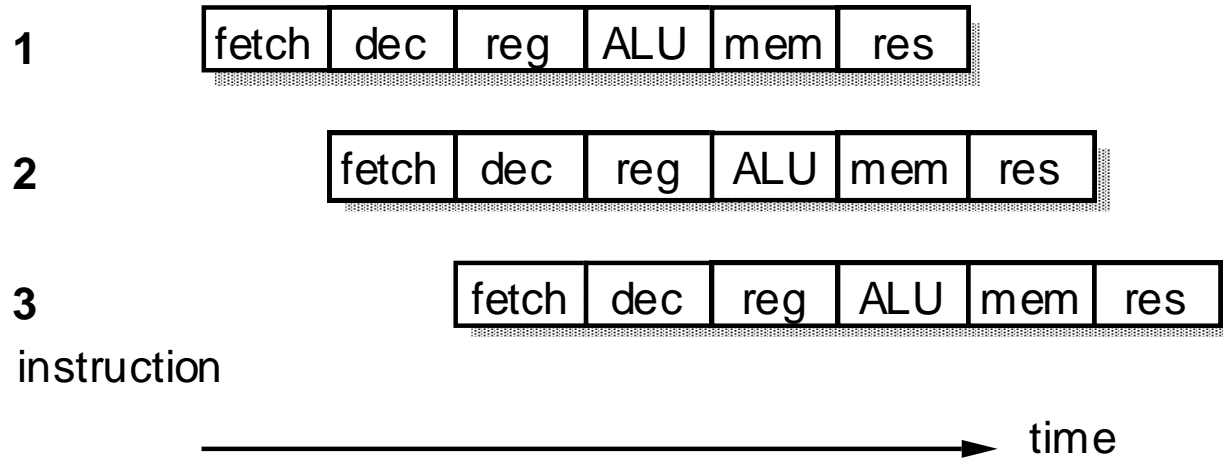
# Pipelined Design

- Prevalent in today's processor implementations
- More pipeline stage
  - Improve throughput
  - Help to increase clock frequency



# Pipelines (1)

# Pipelines (2)



- **fetch**: fetch the instruction from memory
- **dec**: decode it to see what sort of instruction it is
- **reg**: access any operands that may be required from the register bank
- **ALU**: combine the operands to form the result or a memory address
- **mem**: access memory for a data operand, if necessary
- **res**: write the result back to the register bank

# More Pipeline stages, Better Performance?

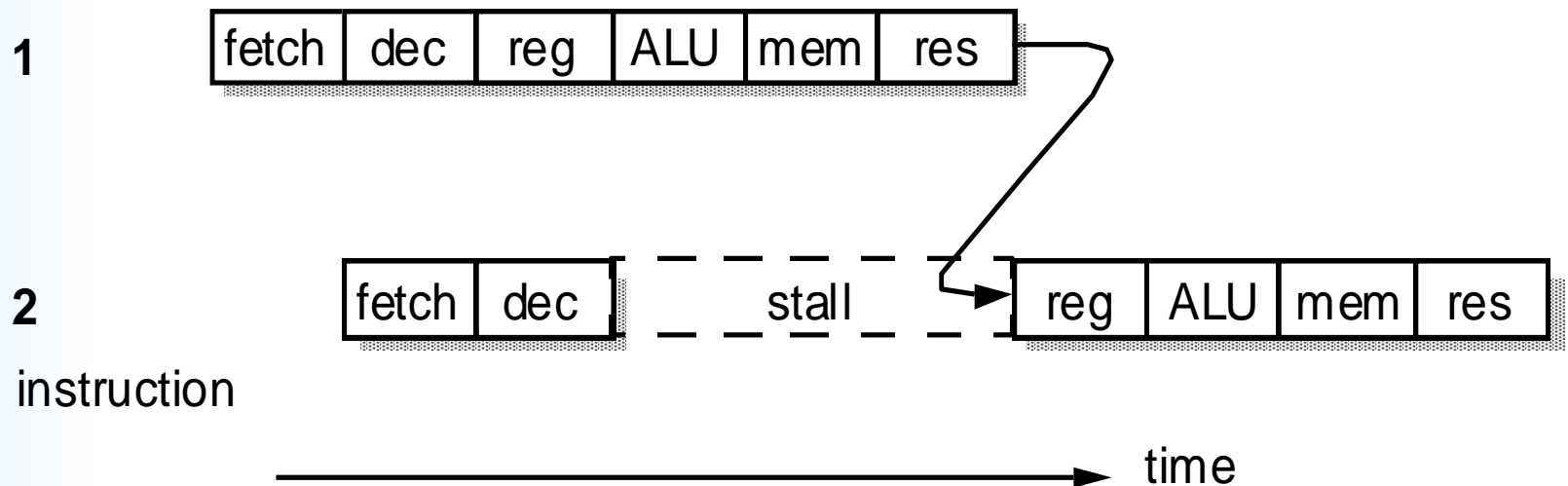
- Pentium 3: 10
- Pentium 4 (Old): 20
- Pentium 4 (Prescott): 31
- Next-Generation Micro-Architecture (NGMA): 14

# Pipelines Hazards

Ex:

add **r1**, r2, #10 (write r1)

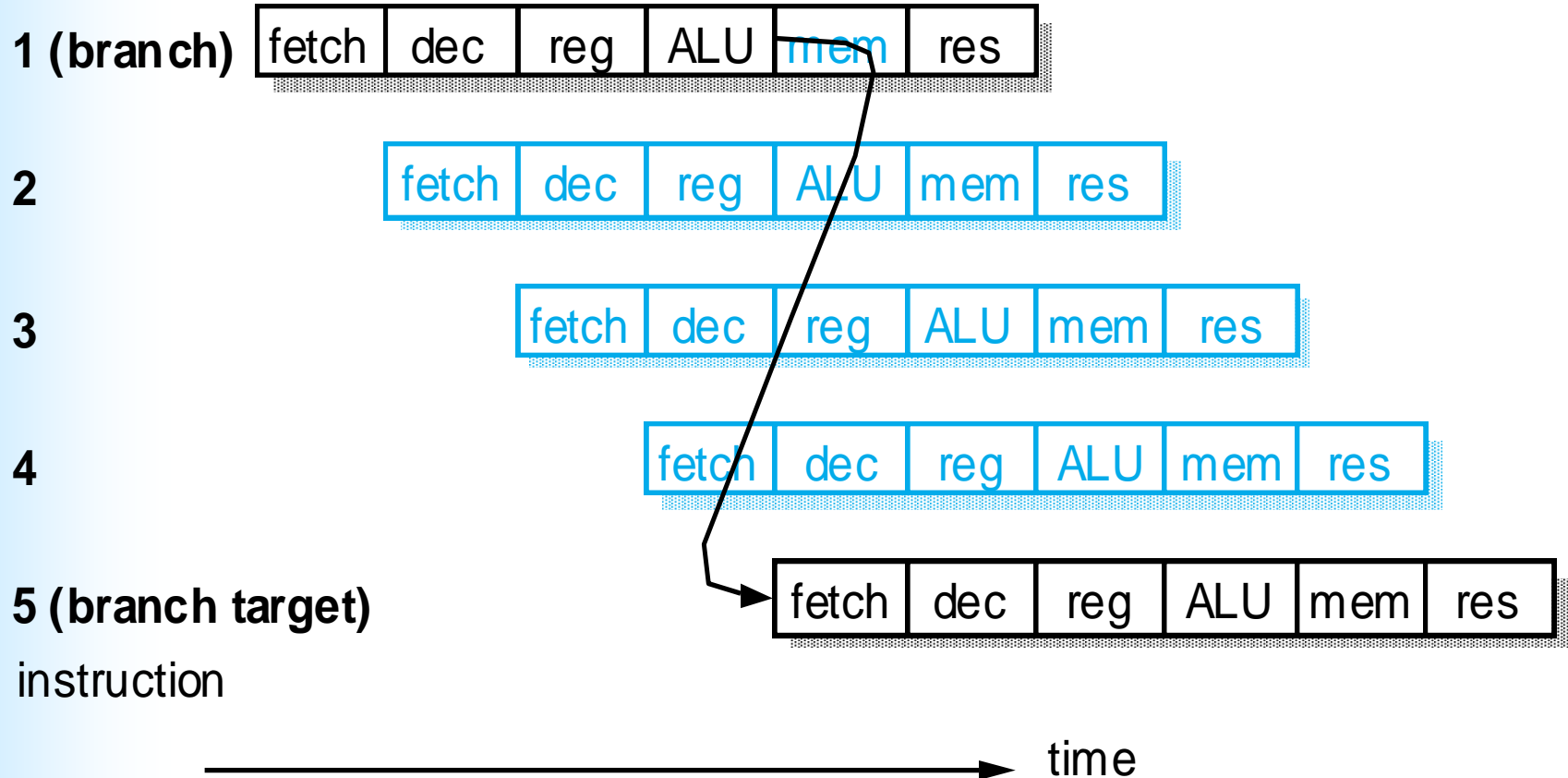
sub r3, **r1**, #20 (use r1)



**Read-after-write pipeline hazard**



# Pipelined Branch Behavior



**Wrong instructions in pipeline need to be flushed (thrown away)**

# Solutions

- Stall pipeline until branch resolved
- Branch prediction
  - Mis-prediction will pay a big penalty
- Q: May we remove branch instruction?
  - **Conditional execution**: operations based on the value of a Boolean source operand
  - **drawback**
    - Affect instruction cache
    - Increase the critical path length