

嵌入式微处理器系统

崔光佐

普适计算与应用实验室

北京大学现代教育技术中心

www.uclab.org

第二篇 ARM微处理器体系机构

第一讲 ARM微处理器体系结构概述

2004.2.21

内容摘要

- ARM简介
- ARM系统结构
 - 系统结构版本演变
 - 编程模型
 - 指令集
 - ARM扩展
- 协处理器接口
- 处理器核
 - ARM7, ARM9, StrongARM
- AMBA总线
- 对操作系统支持
 - 存储系统
 - 堆栈和子程序
 - ARM 软件开发



ARM简介

- **Advances RISC Machines (称为ARM) 由 Acorn, Apple 和 VLSI公司1990年11月联合建立**
- **ARM是16/32位嵌入式RISC处理器提供商的领导企业**
- **该公司主要提供高性能, 低价位, 低功耗的RISC处理器、外围设备,和系统芯片设计给重要的国际电子企业。**
- **ARM也对完整系统的开发提供广泛的支持**
- **32位的load/store型的RISC处理器**



ARM 系统结构

- **系统结构版本**
 - **Version 1 (已经不用)**
 - 基本的数据处理
 - 字节, 字以及多字的load/store
 - 软件中断
 - 26 位地址总线
 - **Version 2 (已经不用)**
 - 乘法 & 乘加操作
 - 协处理器支持
 - 线索同步的原子指令
 - 26 位地址总线



ARM architecture

■ 系统结构版本 (延续)

■ Version 3

- 32 为地址总线
- 增加了CPSR, SPSR
 - 增加了MRS, MSR. 修改异常处理
- 增加了‘数据失效模式’和‘无定义模式’

■ Version 4

- 半字传送
- 引入THUMB处理器状态
- 增加了‘特权模式’给操作系统
- PC距离当前的指令为两个字距离
 - ‘PC+8’
- 第一个正式结构



ARM 系统结构

■ 系统结构版本 (延续)

■ Version 5

- 改进了 ARM/THUMB交替工作
- 增加CLZ指令提高整数除法的效率
- 增加了软件断点
- 对协处理器更多的支持

■ Version 6

- 增加了SIMD指令
- 增加了并行处理的功能

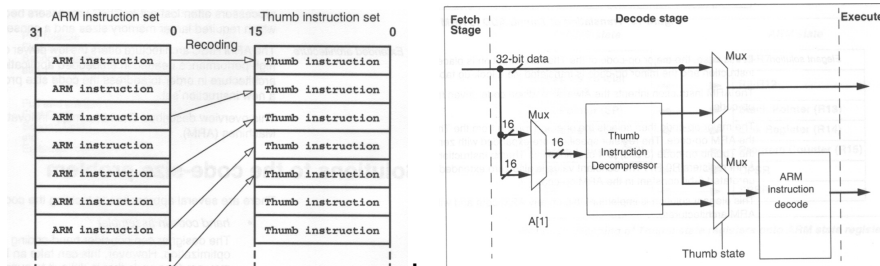


ARM系统结构

■ 结构演变

■ THUMB (symbol as a 'T')

- THUMB指令集: 32位指令的16位重新编码



- 代码量小 (压缩到 40 %)



ARM系统结构

■ 系统结构演变 (延续)

■ 长乘法指令 ('M')

- $32 \times 32 = 64$ bit. 提供全64位结果

■ 增强的DSP指令 ('E')

- 在原有的ARM指令集中增加DSP指令
- 饱和运算
- 64位传送
- V5中第一次引入

■ 处理器核的变化

- D: 片上调试. 暂停功能
- I: 嵌入的ICE. 在片断点



ARM系统结构

■ ARM编程模型

- 32位RISC处理器 (32-bit data & address bus)
- Big and Little Endian operating modes
- 快速中断响应(实时应用)
- 虚拟存续系统支持
- 高级语言支持
- 简单但强大的指令集
 - RISC优势 + CISC优势

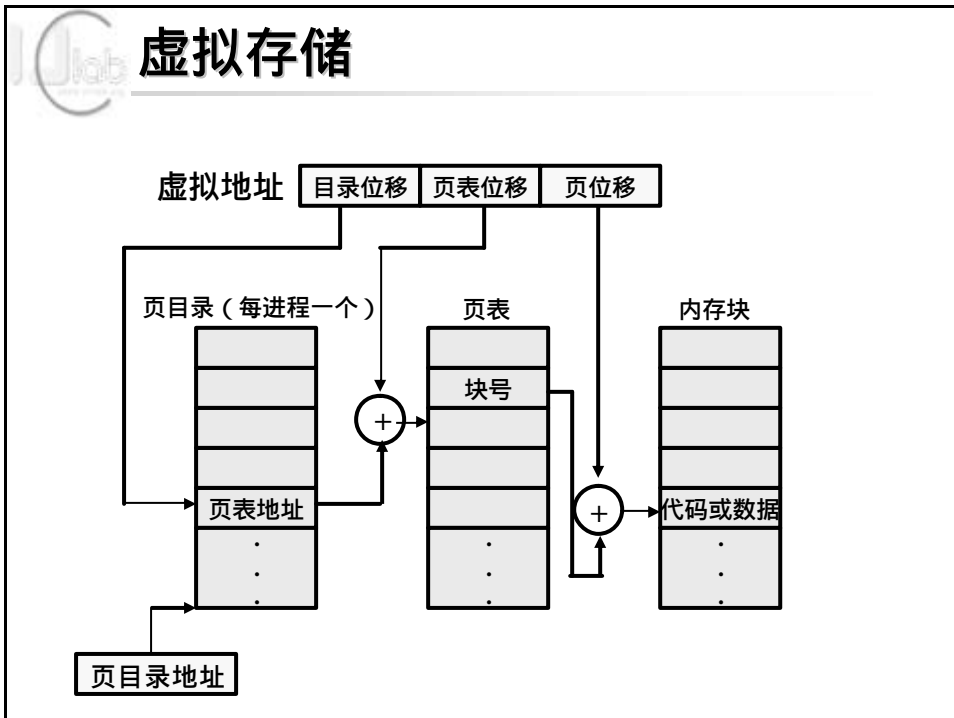


编程模型

- Enianism (configured by input signal)
 - Big Endian
- 最高有效字节在最低地址
- 字按照最高有效字节寻址
- Little Endian
- 最低有效字节存于最低地址
- 字按照最低有效字节的地址寻址

Higher Address	31	24 23	16 15	8 7	0	Word Address
	11	10	9	8		8
	7	6	5	4		4
Lower Address	3	2	1	0		0

Higher Address	31	24 23	16 15	8 7	0	Word Address
	8	9	10	11		8
	4	5	6	7		4
Lower Address	0	1	2	3		0



编程模型

■ 操作模式 (由软件配置)

- User mode (usr)
 - 一般用户程序执行状态
- FIQ mode (fiq)
 - 支持数据传送和通道处理
- IRQ mode (irq)
 - 通常的中断处理
- Supervisor mode (svc)
 - 操作系统保护的模式
- Abort mode (abt)
 - 数据或指令预取失效后进入的状态
- Undefined mode (und)
 - 执行未定义的指令时进入的模式



编程模型

■ 寄存器 (延续)

User32	Fiq32	Supervisor32	Abort32	IRQ32	Undefined32
R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7
R8	R8_fiq	R8	R8	R8	R8
R9	R9_fiq	R9	R9	R9	R9
R10	R10_fiq	R10	R10	R10	R10
R11	R11_fiq	R11	R11	R11	R11
R12	R12_fiq	R12	R12	R12	R12
R13	R13_fiq	R13_svc	R13_abt	R13_irq	R13_und
R14	R14_fiq	R14_svc	R14_abt	R14_irq	R14_und
R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)	R15(PC)

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_fiq	SPSR_svc	SPSR_abt	SPSR_irq	SPSR_und

- **R13: 堆栈指针 (与通常的意义一样)**
 - 每个处理器模式具有不同的堆栈指针
- **R14: 链接寄存器**
- **R15: 程序指针**



编程模型

■ 寄存器

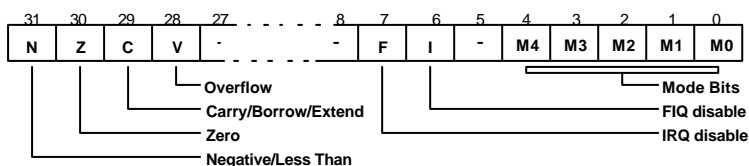
- 37 个寄存器
 - ? 31 个通用32位寄存器
 - ? 6 各状态寄存器
- 16个通用寄存器和一个或2个状态寄存器在任何时候都可见
- 具体可见的寄存器依赖处理器的模式
- 其它的寄存器(分块的寄存器)分成不同的组以支持 IRQ, FIQ, Supervisor, Abort 和未定义模式处理
- R0 至 R15可直接访问
- R0 to R14 为通用寄存器
- R15保存程序计数器 (PC)
- CPSR - Current Program Status Register contains condition code flags and the current mode bits
- 5 SPSRs (Saved Program Status Registers) which are loaded with CPSR when an exceptions occurs



编程模型

■ 处理器状态寄存器 (CPSR/SPSR)

- N, Z, C 和 V 为条件码标示
 - 这些位根据算术逻辑操作的结果而设置
 - 被所有的指令测试来确定指令是否被执行
 - N : Negative. Z : Zero. C : Carry. V : oVerflow
- I 和 F 位是中断禁止位
- M0, M1, M2, M3 和M4为模式标志位



ARM中的执行模式

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

Table 2: The Mode Bits



ARM中的执行模式

Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	-- reserved --	--
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ



编程模式

■ 异常

■ 打断程序的正常执行

- 处理来自外围设备的中断
- 确保正在流水线中执行的指令

■ 异常类型

- FIQ (Fast Interrupt reQuest)
 - 外部事件产生, 通过nFIQ输入为低电平
 - 数据或通道的快速处理
- IRQ(Interrupt ReQuest)
 - 常规中断, nIRQ输入为低电平有效
- ABORT
 - 外部信号ABORT引起
 - 暗示当前的存储器访问不能正常完成
- 软件中断
 - 由软件中断指令引起 (SWI)
 - 进入超级管理模式
 - 通常请求一个特殊的超级管理功能. 支持操作系统



FIQ

When a FIQ is detected, ARM7 performs the following:

- (1) Saves the address of the next instruction to be executed plus 4 in R14_fiq; saves CPSR in SPSR_fiq**
- (2) Forces M[4:0]=10001 (FIQ mode) and sets the F and I bits in the CPSR**
- (3) Forces the PC to fetch the next instruction from address 0x1C**

To return normally from FIQ, use SUBS PC, R14_fiq,#4 which will restore both the PC (from R14) and the CPSR (from SPSR_fiq) and resume execution of the interrupted code.



IRQ

When an IRQ is detected, ARM7 performs the following:

- (1) Saves the address of the next instruction to be executed plus 4 in R14_irq; saves CPSR in SPSR_irq**
- (2) Forces M[4:0]=10010 (IRQ mode) and sets the I bit in the CPSR**
- (3) Forces the PC to fetch the next instruction from address 0x18**

To return normally from IRQ, use SUBS PC,R14_irq,#4 which will restore both the PC and the CPSR and resume execution of the interrupted code.



编程模型

■ 异常 (继续)

■ 异常的类型 (继续)

- 无定义的指令自陷
 - When the ARM comes across an instruction which it cannot handle it offers it to any coprocessors which may be present
 - If a coprocessor can perform this instruction but is busy at that time, ARM will wait until the coprocessor is ready or until an interrupt occurs
 - If no coprocessor can handle the instruction then ARM will take the undefined instruction trap

■ Exception Priorities

- (1) Reset (highest priority)
- (2) Data abort
- (3) FIQ
- (4) IRQ
- (5) Prefetch abort
- (6) Undefined Instruction, Software interrupt (lowest priority)



- 正在为大家建账户，下周2下午2：00建完。利用该工具交作业
- 网址：作业情况
- 用户号：学号
- Pw:学号

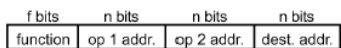


ARM系统结构

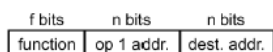
■ 指令集

■ 指令格式

- 3地址格式
 - ARM状态使用



- 2地址格式
 - ARM和 THUMB 状态



ARM指令系统

■ 条件执行

- 所有的ARM指令都是条件执行
- 指令是否执行依赖于CPSR中的N, Z, C和V标志位
- 所有的THUMB指令都解压成 ‘Always’ 条件指令
- 指令中的条件域



- 0000 = EQ - Z set (equal)
- 0001 = NE - Z clear (not equal)
- 0010 = CS - C set (unsigned higher or same)
- 0011 = CC - C clear (unsigned lower)
- 0100 = MI - N set (negative)
- 0101 = PL - N clear (positive or zero)
- 0110 = VS - V set (overflow)
- 0111 = VC - V clear (no overflow)
- 1000 = HI - C set and Z clear (unsigned higher)
- 1001 = LS - C clear or Z set (unsigned lower or same)
- 1010 = GE - N set and V set, or N clear and V clear (greater or equal)
- 1011 = LT - N set and V clear, or N clear and V set (less than)
- 1100 = GT - Z clear, and either N set and V set, or N clear and V clear (greater than)
- 1101 = LE - Z set, or N set and V clear, or N clear and V set (less than or equal)
- 1110 = AL - always
- 1111 = NV - never



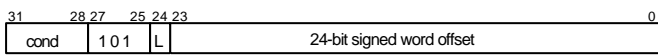
转移指令

- 两种方法：
 - 转移指令
 - 直接写PC
- 转移指令
 - B跳转指令
 - BL带返回的跳转指令
 - BLX带返回和状态切换的跳转指令
 - BX待状态切换的跳转指令

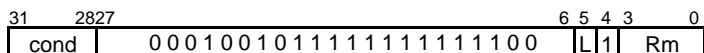


指令集（续）

- 条件指令
 - 转移与带链接的转移
 - 转移到目标指令
 - 保存当前的PC到链接寄存器 (使用 ‘L’位)



- 转移与转换
 - 转移到目标指令并转换指令集
 - Rm[0] == 1: 后面的指令为THUMB指令集.
 - Rm[0] == 0: 后面的指令为ARM指令集.





指令集（续）

- 条件指令
- 条件转移指令
 - 转移与带链接的转移
 - 转移到目标指令
 - 保存当前的PC到链接寄存器 (使用 ‘L’位)

指令操作的伪代码

IF ConditionalPassed(cond) then

if L==1 then

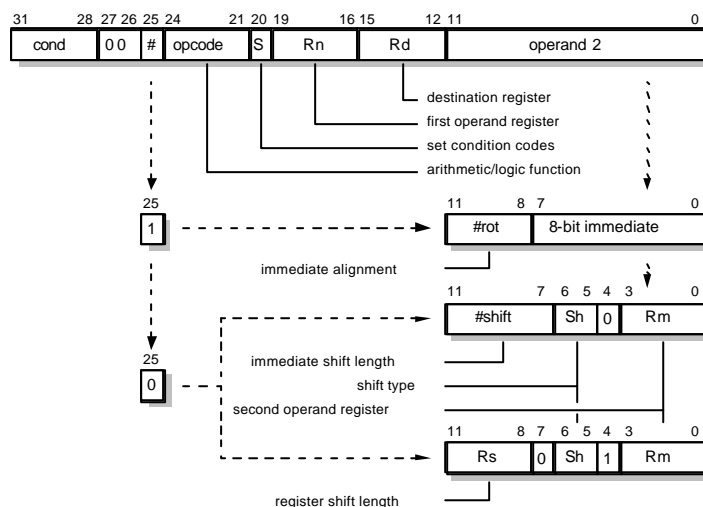
LR=address of the instruction after the branch instruction

PC=PC+(SignExtend(signimm24)<<2)



指令集（续）

■ 数据处理指令





指令集(续)

■ 数据处理指令

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	$Rd := Rn \text{ AND } Op2$
0001	EOR	Logical bit-wise exclusive OR	$Rd := Rn \text{ EOR } Op2$
0010	SUB	Subtract	$Rd := Rn - Op2$
0011	RSB	Reverse subtract	$Rd := Op2 - Rn$
0100	ADD	Add	$Rd := Rn + Op2$
0101	ADC	Add with carry	$Rd := Rn + Op2 + C$
0110	SBC	Subtract with carry	$Rd := Rn - Op2 + C - 1$
0111	RSC	Reverse subtract with carry	$Rd := Op2 - Rn + C - 1$
1000	TST	Test	Scc on $Rn \text{ AND } Op2$
1001	TEQ	Test equivalence	Scc on $Rn \text{ EOR } Op2$
1010	CMP	Compare	Scc on $Rn - Op2$
1011	CMN	Compare negated	Scc on $Rn + Op2$
1100	ORR	Logical bit-wise OR	$Rd := Rn \text{ OR } Op2$
1101	MOV	Move	$Rd := Op2$
1110	BIC	Bit clear	$Rd := Rn \text{ AND NOT } Op2$
1111	MVN	Move negated	$Rd := \text{NOT } Op2$

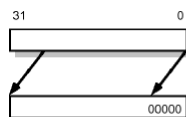


指令集 (续)

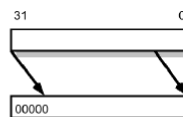
■ 数据处理指令(续)

■ 移位操作

- 在任何数据处理指令中，第二个寄存器操作数都可以使用移位操作。
- 逻辑移位




LSL #5



LSR #5

- LSL: 逻辑左移0到31位
 - 字中左移的空位用0填充。
- LSR: 逻辑右移0到31位
 - 字中右移空出的最高有效位用零填充



指令集（续）

- 数据处理指令（续）
 - 移位指令(续)
 - 算术移位
 - ASL: = LSL
 - ASR: Arithmetic shift left
 - Sign extend the shifting bits

31

0

0

00000 0

ASR #5, positive operand

31

0

1

11111 1

ASR #5, negative operand

31

0

ROR #5

31


0

C

C

RRX

- Rotation: ROR, RRX



指令集（续）

- 乘法指令
 - 两个32位寄存器相乘
 - 需要多个周期和高的功耗
 - 尽量转换成其它指令
 - Ex) $b = a * 5 : b = a + a << 2$

31	28 27	24 23	21 20 19	16 15	12 11	8 7	4 3	0
cond	0 0 0 0	mul	S	Rd/RdHi	Rn/RdLo	Rs	1 0 0 1	Rm

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	$Rd := (Rm * Rs) [31:0]$
001	MLA	Multiply-accumulate (32-bit result)	$Rd := (Rm * Rs + Rn) [31:0]$
100	UMULL	Unsigned multiply long	$RdHi: RdLo := Rm * Rs$
101	UMLAL	Unsigned multiply-accumulate long	$RdHi: RdLo += Rm * Rs$
110	SMULL	Signed multiply long	$RdHi: RdLo := Rm * Rs$
111	SMLAL	Signed multiply-accumulate long	$RdHi: RdLo += Rm * Rs$



指令集（续）

■ 数据传送指令

■ 单个数据传送 (LDR, STR)

- 单个字(32bit), 半字(16 bit) 和字节(8 bit)传送
- 寻址
 - 寄存器偏移量
 - 地址 = 基址寄存器内容 \pm 偏移寄存器内容
 - 立即偏移量
 - 地址 = 基址寄存器 \pm 立即常数
 - 后变址: 使用后改变地址
 - 前基址: 使用前改变地址
- 基址写回
 - 使能时, 更新基址寄存器



指令集（续）

■ 数据传送指令（续）

■ 多个数据转移 (LDM, STM)

- 装入(LDM)或保存 (STM) 可见寄存器的任意一个子集
- 应用
 - 堆栈: 维护堆栈, 生序次序或降序次序
 - 上下文切换: 保存或恢复工作寄存器
 - 块拷贝: 在内存中移动大块数据
- 寻址
 - 前/后变址
 - 自动增量/减量
 - 写回基址寄存器
- 特殊位
 - PSR 强制用户位



指令集（续）

■ 数据传输（续）

■ 单数据交换 (SWAP)

- 一个寄存器和外部主存之间交换一个字节或字
- 锁定在一起的一个读内存和写内存操作
 - 原子指令
 - 执行中不能中断
 - 外部存储管理单元被‘LOCK’输出信号锁定
- 应用
 - 多个线索程序之间同步 (操作系统支持)
 - 锁技术
 - 信号灯技术



指令集（续）

■ PSR指令(MRS, MSR)

■ MRS和MSR指令属于数据处理指令

■ 这些指令允许访问CPSR和SPSR寄存器：

- MRS指令可将CPSR或SPSR_<mode>的内容保存到通用寄存器
- MSR可将通用寄存器的内容传送到CPSR或SPSR_<mode>寄存器



指令集（续）

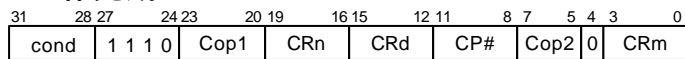
■ 协处理器指令

■ 协处理器

- 扩展指令集的通用机制
- 例如：象 MMU & cache 等的系统控制、FPU
- 寄存器
 - 协处理器私有
 - ARM控制数据流
 - 协处理器只是涉及数据处理和存储转移操作

■ 协处理器数据操作(CDP)

- 该类指令用来告诉协处理器执行某些内部的操作
- 这些操作结果不返回给ARM,而且ARM也不等待操作完成



指令集（续）

■ 协处理器指令（续）

■ 协处理器数据传送 (LDC, STC)

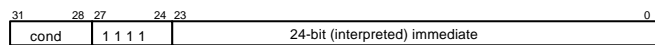
- 直接Load (LDC)或store (STC) 协处理器寄存器的子集到内存
- ARM负责提供内存地址，协处理器负责提供或接收数据并控制数据的传输量

■ 协处理器寄存器传输 (MRC, MCR)

- ARM和一个协处理器之间直接传输信息

■ 软件中断指令

- 用来进入超级管理模式
- 该指令导致软件中断自陷,引起模式改变



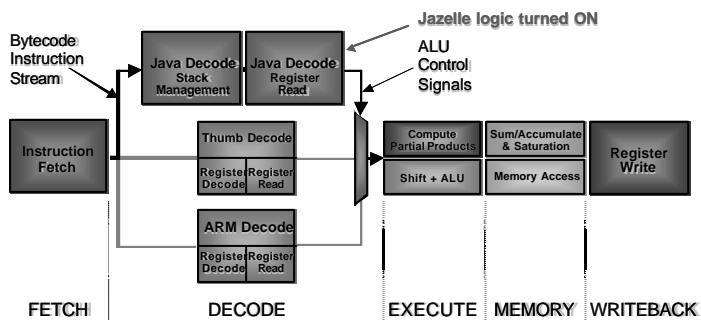


ARM系统结构

■ ARM 扩展

■ Jazelle™ : ARM的Java扩展(‘J’符号表示)

- 指令扩展(Java状态, 而非协处理器)
- ARM 流水线实现
- 堆栈到寄存器的动态映射

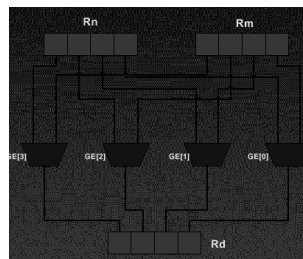


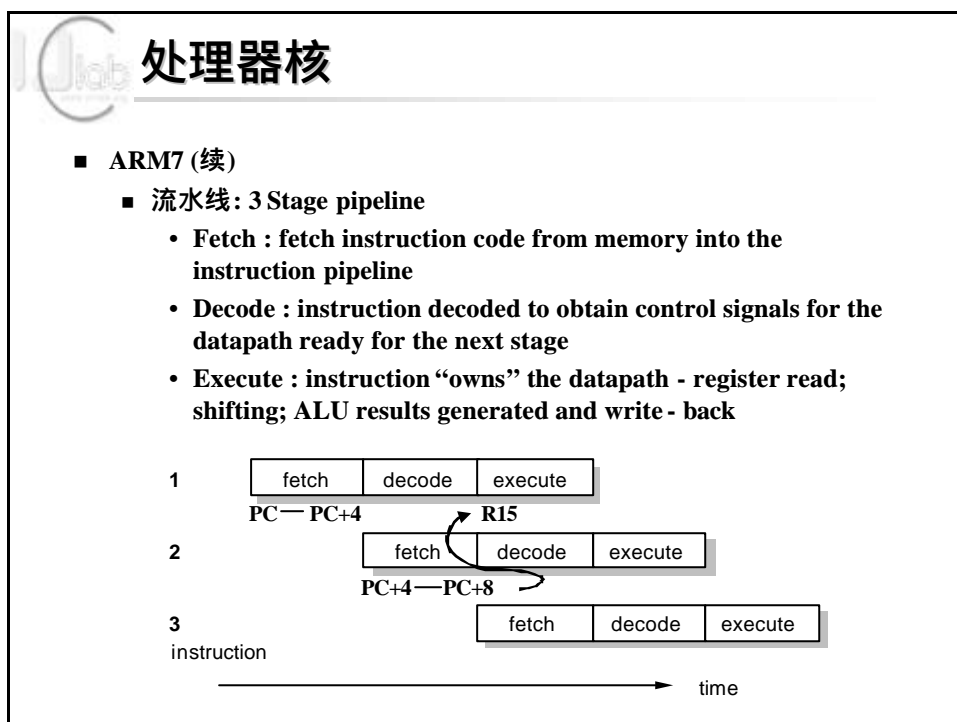
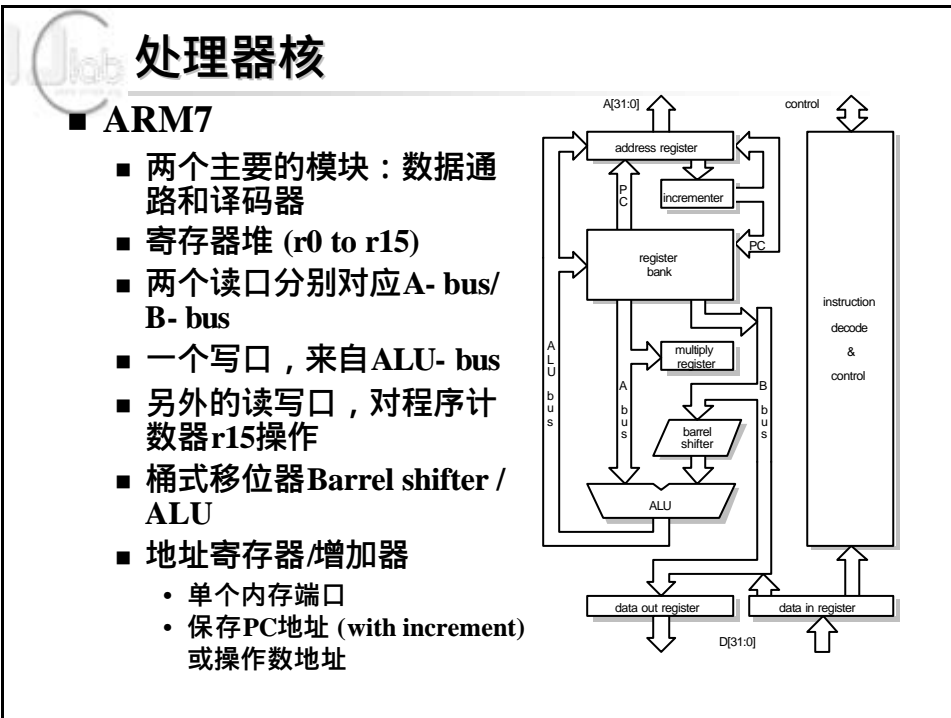
ARM系统结构

■ ARM 扩展 (续)

■ ARM version 6

- 加强内存管理
- 并行处理
 - 增加了新的同步指令 (LDREX, STREX)
- 增强例外处理
 - PSR中增加新的bit
- 混合的内存存放方式
- 媒体扩展
 - ARM SIMD
 - (16bit 2 way and 8 bit 4 way)
 - FFT, MPEG4
 - Saturation, ...





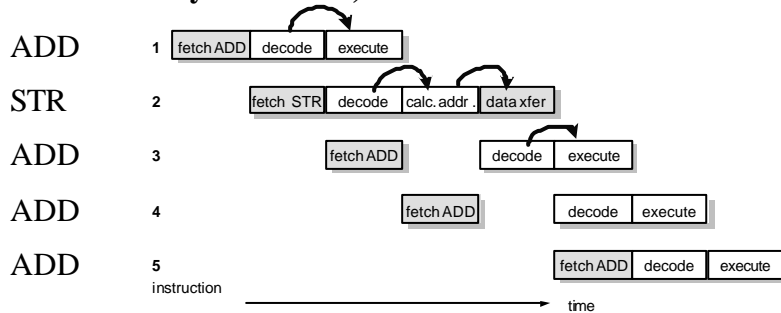


Processor Cores

■ ARM7(cont'd)

■ Multi-cycle operation

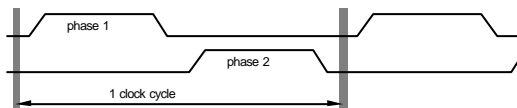
- Single cycle throughput for almost simple data processing instruction
- Multi-cycle for mul, load/store



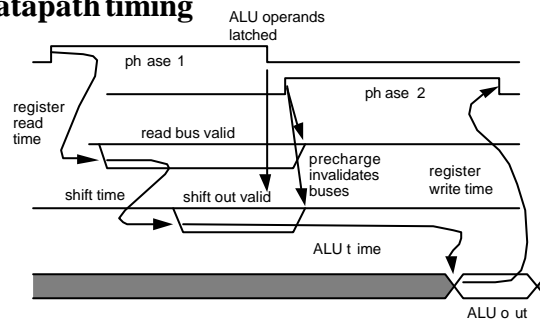
Processor Cores

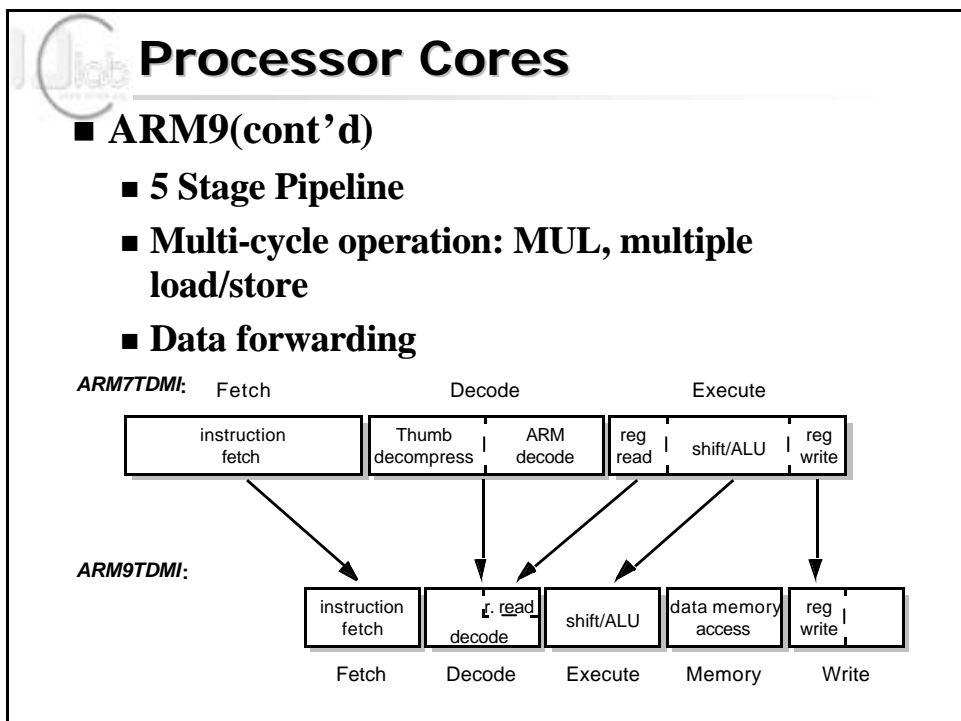
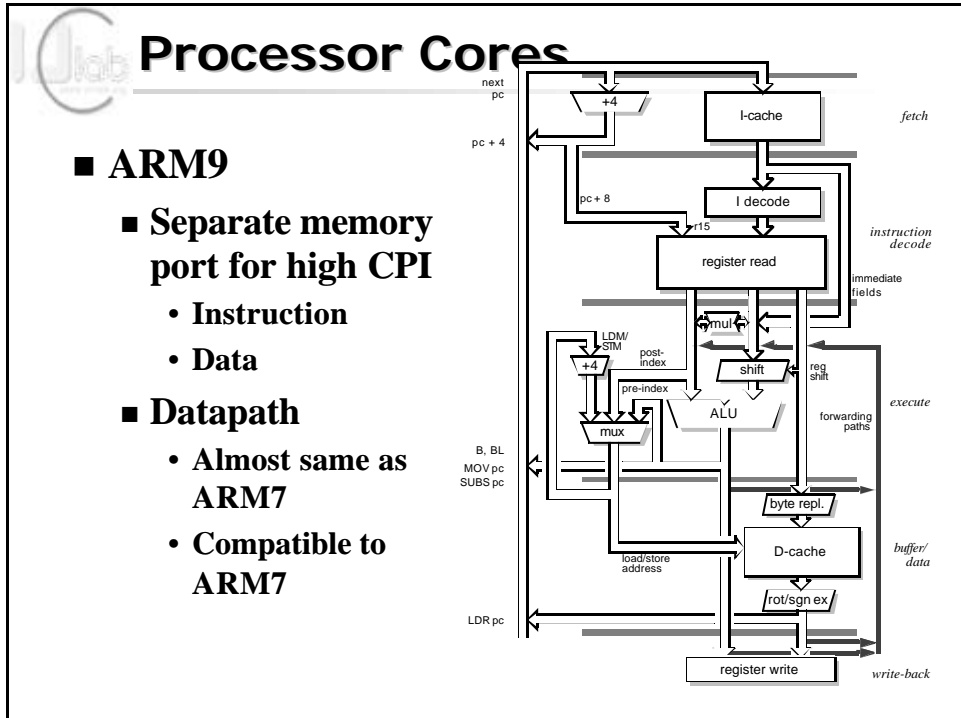
■ ARM7(cont'd)

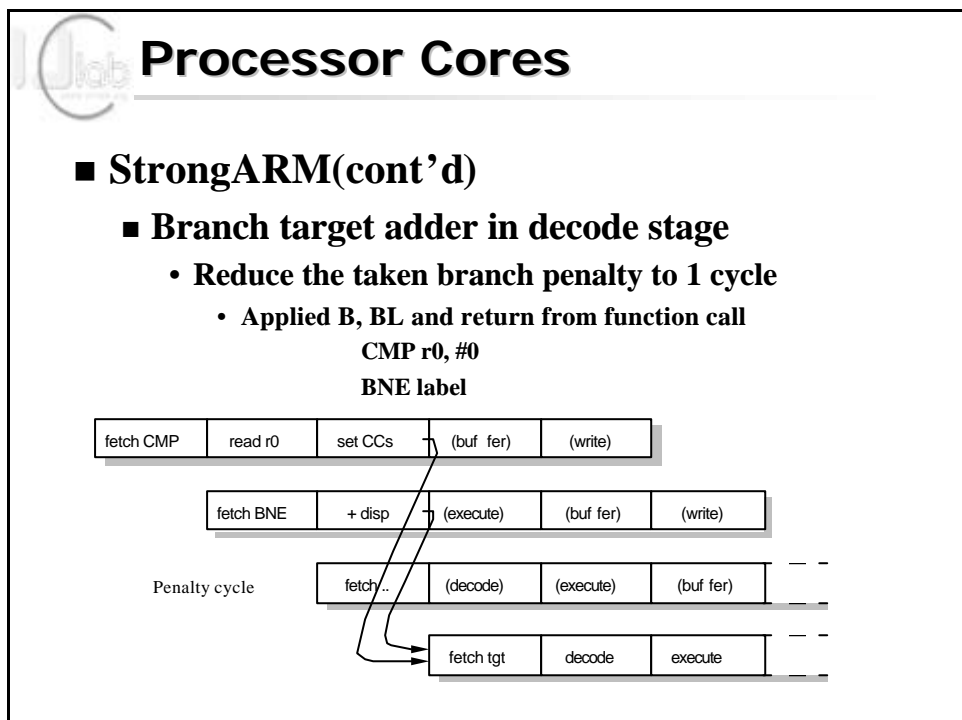
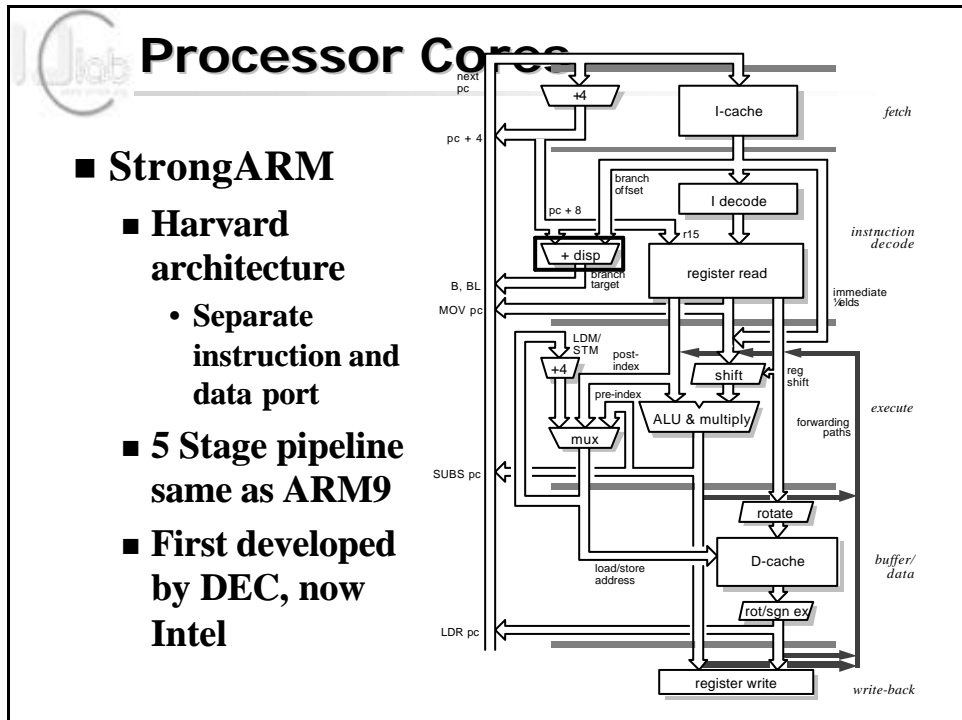
■ 2 Phase Non-overlapping clocking scheme



■ Datapath timing









Processor Cores

■ StrongARM(cont'd)

■ Multiply implementation

	Memory port	Multiplier	Branch adder
ARM7	1	8 bit	x
ARM9	2	8 bit	x
StrongARM	2	12 bit	0

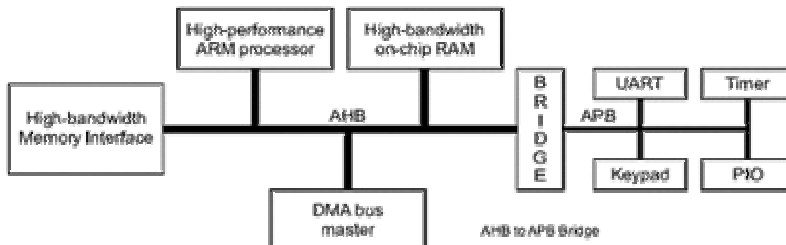
- Reduce the issue latency of MUL to 1 ~3 cycle
 - Compared to ARM7, ARM9 (1~4 cycle)



AMBA

■ Advanced Microcontroller Bus Architecture

- Standard of on-chip communication between different macrocells for high performance embedded system design
- Hierarchical Bus architecture





AMBA

■ AMBA buses

■ AHB(Advanced High Performance Bus)

- Connect between high-performance system modules

■ ASB(Advanced System Bus)

- Subset of AHB

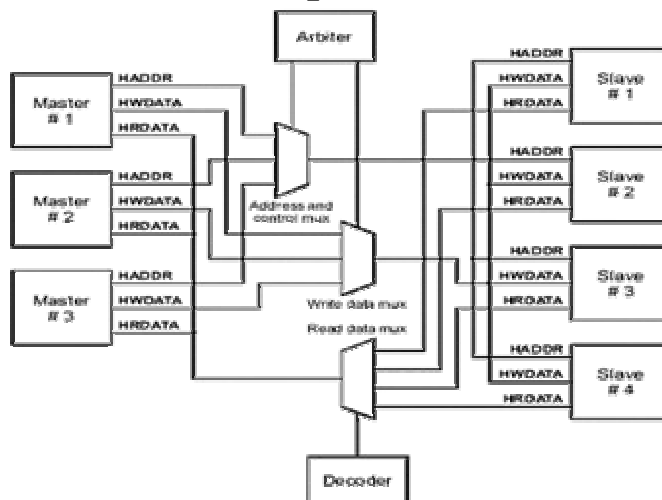
■ APB(Advanced Peripheral Bus)

- Simple interface for low-performance peripherals



AMBA

■ AMBA AHB component (cont'd)



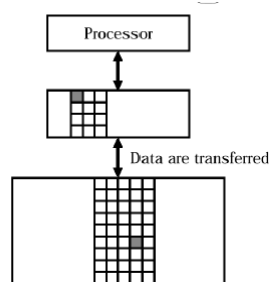
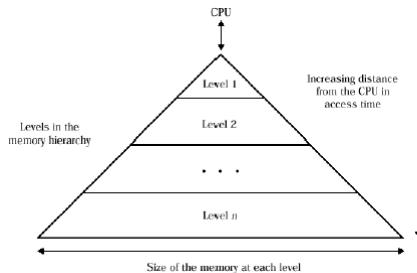


Operating System Support

■ Memory System

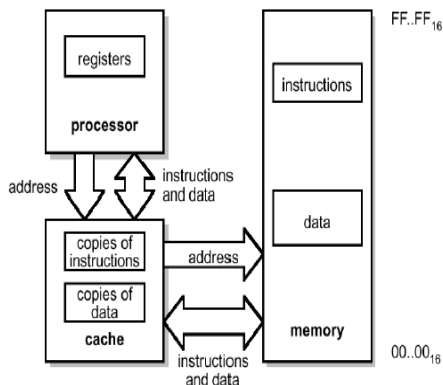
■ Memory hierarchy

- Cache system
 - Temporal locality
 - Spatial locality

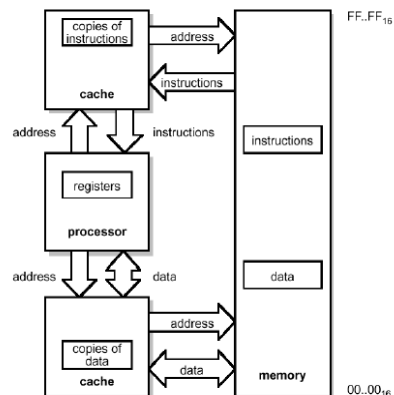


Cache system (cont'd)

■ Single cache shared between instruction and data



• Separate data and instruction cache





Cache system (cont'd)

■ Write strategy

■ Write- through

- All write are passed to main memory immediately
- If there is a hit, the cache is updated to hold new value
- Processor slow down to main memory speed during write

■ Write- through with buffered write

- Use a buffer to hold data to write back to main memory
- Processor only slowed down to write buffer speed (which is fast)
- Write buffer transfers data to main memory (slowly), processor continues its tasks

■ Copy- back

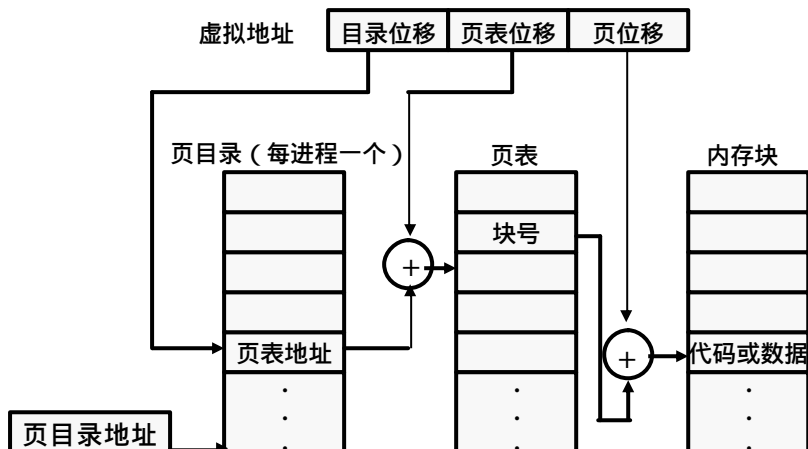
- Write operation updates the cache, but not main memory
- Cache remember that it is different from main memory via a dirty bit
- It is copied back to main memory only when the cache line is used by new data



Operating System Support

■ Memory Management Unit

■ Virtual memory system

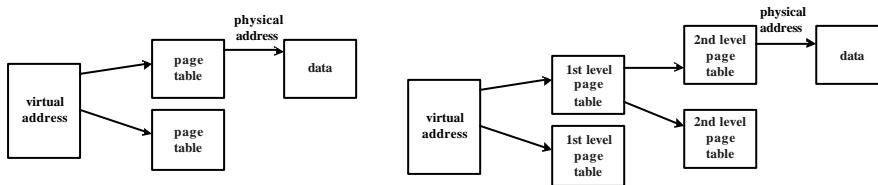




Memory Management Unit (cont'd)

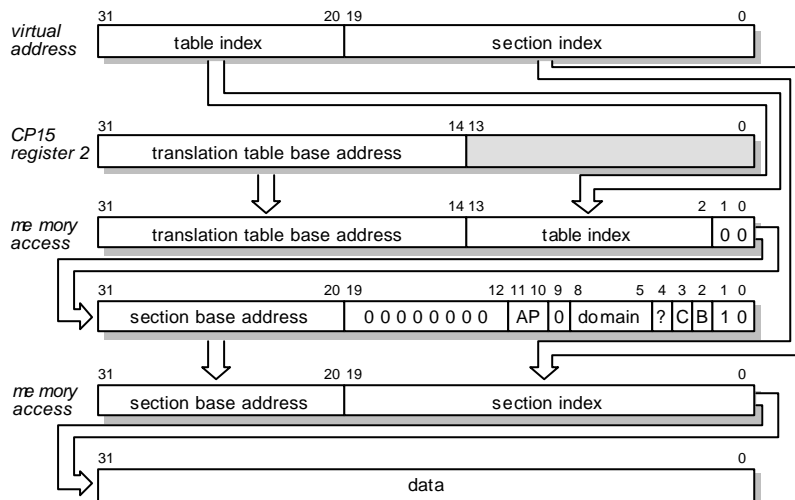
■ ARM MMU

- Translates virtual address to physical address
- Controls memory access permission
- Use 2 level page table with TLB
 - Page: fixed size of chunk of memory
 - TLB: cache of virtual to physical address mapping



Memory Management Unit (cont'd)

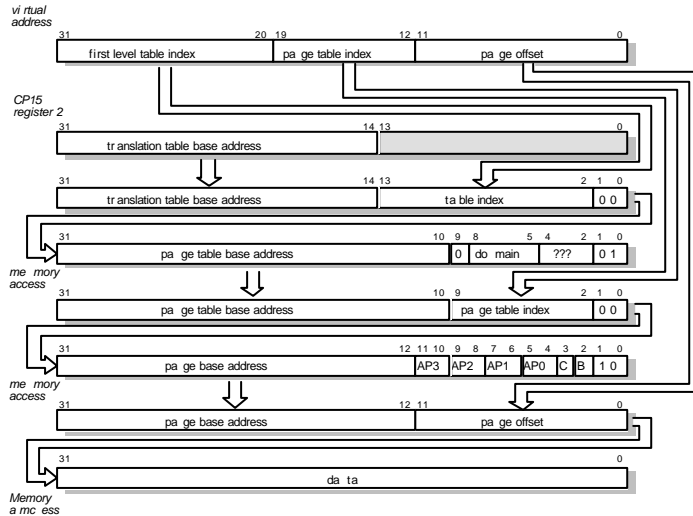
■ Selection translation sequence





Memory Management Unit (cont'd)

■ Small page translation sequence

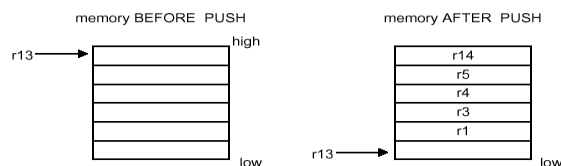


Operating System Support

■ Stack and Subroutine System

■ Idea of stack

- The multiple load/ store instructions can be used to implement last-in- first- out storage called a **STACK**.
- A stack is a portion of main memory used to store data temporarily
- A **PUSH** operation which stores a number of registers onto the stack memory.

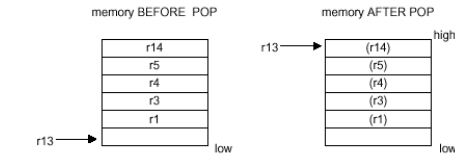




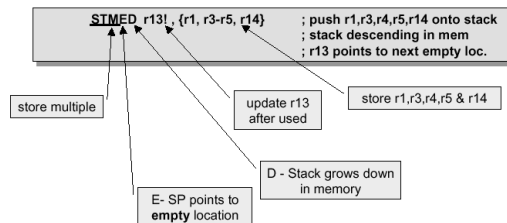
Stack & Subroutine System (cont'd)

■ Stack(cont'd)

■ Pop operation



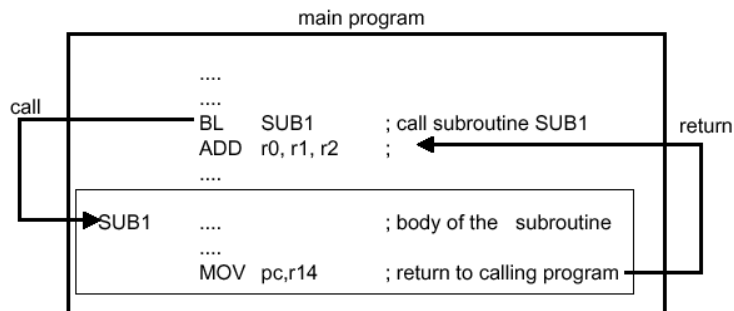
■ Implemented by LDM/STM instruction



Stack & Subroutine System (cont'd)

■ Subroutine

- Subroutines allow you to modularize your code so that they are more reusable.





Stack & Subroutine System (cont'd)

■ Subroutine with saving the context

```
BL    SUB1
.....
SUB1  STMED r13!, {r0-r2, r14}    ; push work & link registers
.....
BL    SUB2                      ; jump to a nested subroutine
...
LDMED r13!, {r0-r2, r14}        ; pop work & link registers
MOV   pc, r14                  ; return to calling program
```

