

CS4803DGC Design Game Console

Spring 2010

Prof. Hyesoon Kim

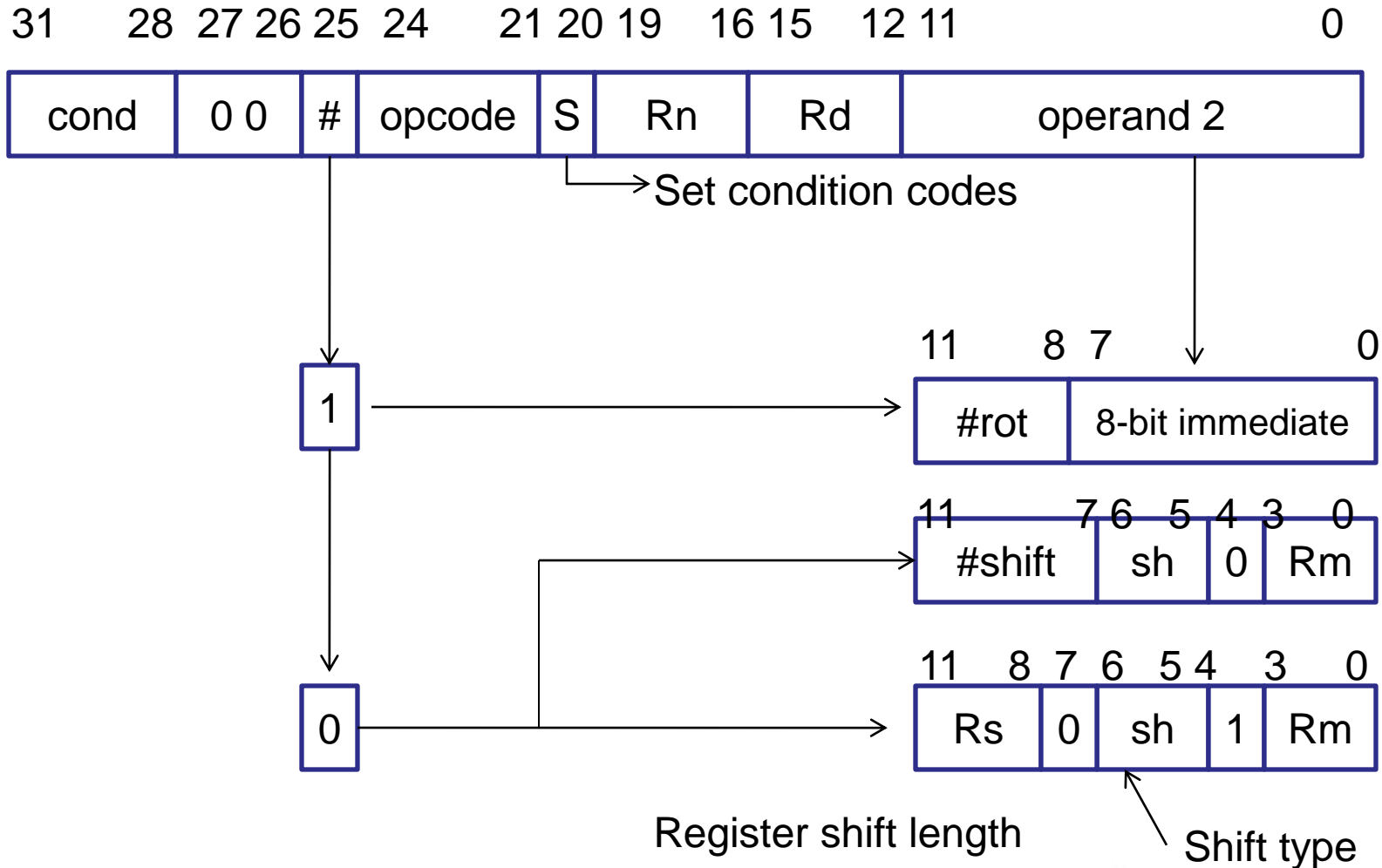


**Georgia
Tech**



College of
Computing

Data Processing Instructions





Immediate Operands

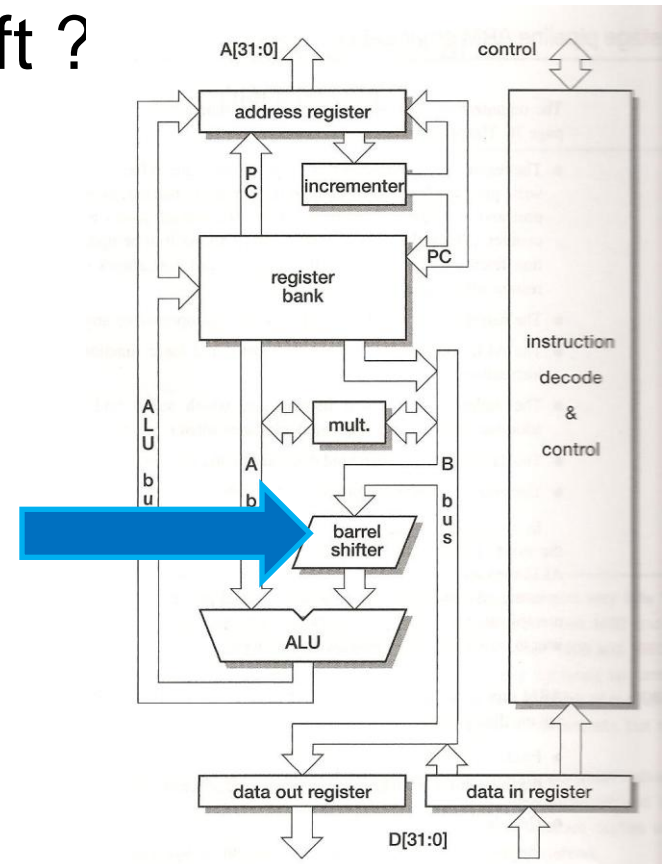
- Using 12 bits how to represent 32-bit immediate value?
- Immediate = $(0 \rightarrow 255) \times 2^{2n}$
 - Where $0 \leq n \leq 12$ (4 bits rotation)
 - 8 bit immediate + 4-bit shift
 - 8 bit + 24 = 32 bit representation



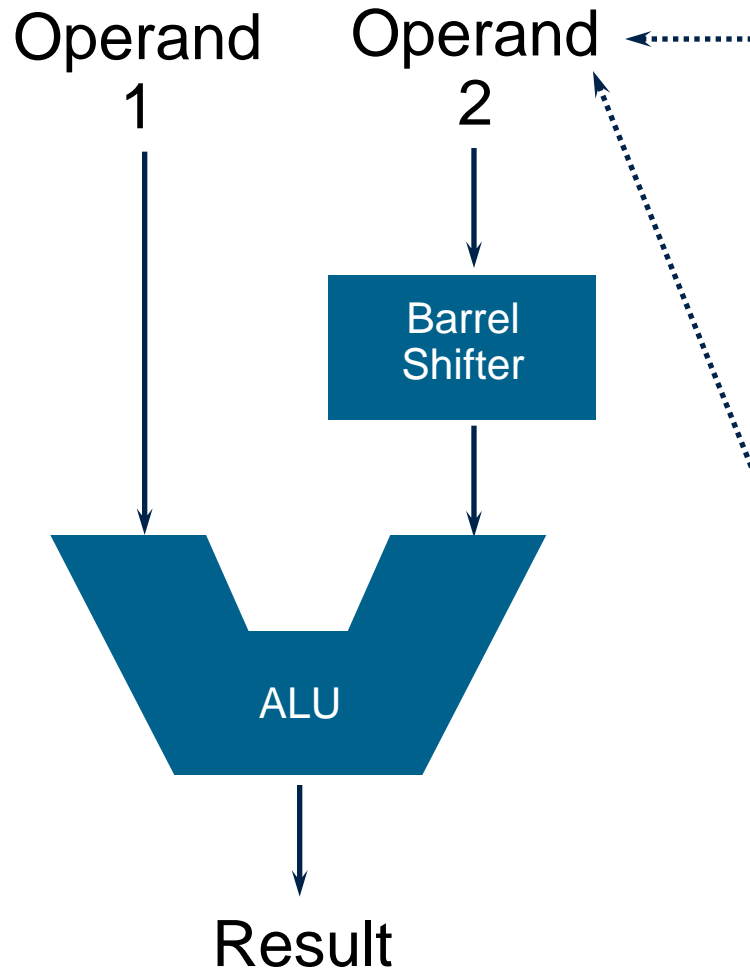
Shifted Register Operations

- ADD r3, r2, r1, LSL #3; r3:=r2+r1*8
- Logical shift vs. Arithmetic shift ?
 - E.g.) b1011 , Carry:1
 - LSL, 1 : b0110
 - LSR, 1: b0101
 - ASL, 1: b0110
 - ASR, 1: b1101
 - ROR, 1: b1101
 - RRX, 1: b1101 carry: 1
Coming from carry bit
- Use register to specify shift
- ADD r5,r5,r3, LSL r2; r5 := r5+r3 x 2^(r2)

Input to the ALU



Using the Barrel Shifter: The Second Operand



Register, optionally with shift operation

- Shift value can be either be:
 - 5 bit unsigned integer
 - Specified in bottom byte of another register.
- Used for multiplication by constant

Immediate value

- 8 bit number, with a range of 0-255.
 - Rotated right through even number of positions
- Allows increased range of 32-bit constants to be loaded directly into registers



ARM Data Processing Instructions

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	Set carry on $Rn \text{ AND } Op2$
1001	TEQ	Test equivalence	Set carry on $Rn \text{ EOR } Op2$
1010	CMP	Compare	Set carry on $Rn - Op2$
1011	CMN	Compare negated	Set carry 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	$Rd := \text{NOT } Op2$



Condition Code Set

- **S** bit (bit 20)
 - 1: condition code is set
 - 0: condition code is unchanged
- **N**: 1: result is negative 0: result is 0 or positive
 - $N = \text{result}[31]$
- **Z**: 1: zero 0: non-zero
- **C**: Carry-out from ALU when the operation is arithmetic
 - ADD, ADC, SUB, SBC, RSB, CMP, CMN
 - Carry out from shifter
- **V**: overflow , non-arithmetic operations do not touch V-bit
 - Only for signed operations



Review: Condition Codes

- The possible condition codes are listed below:
 - Note AL is the default and does not need to be specified

Suffix	Description	Flags tested
EQ	Equal	Z=1
NE	Not equal	Z=0
CS/HS	Unsigned higher or same	C=1
CC/LO	Unsigned lower	C=0
MI	Minus	N=1
PL	Positive or Zero	N=0
VS	Overflow	V=1
VC	No overflow	V=0
HI	Unsigned higher	C=1 & Z=0
LS	Unsigned lower or same	C=0 or Z=1
GE	Greater or equal	N=V
LT	Less than	N!=V
GT	Greater than	Z=0 & N=V
LE	Less than or equal	Z=1 or N!=V
AL	Always	

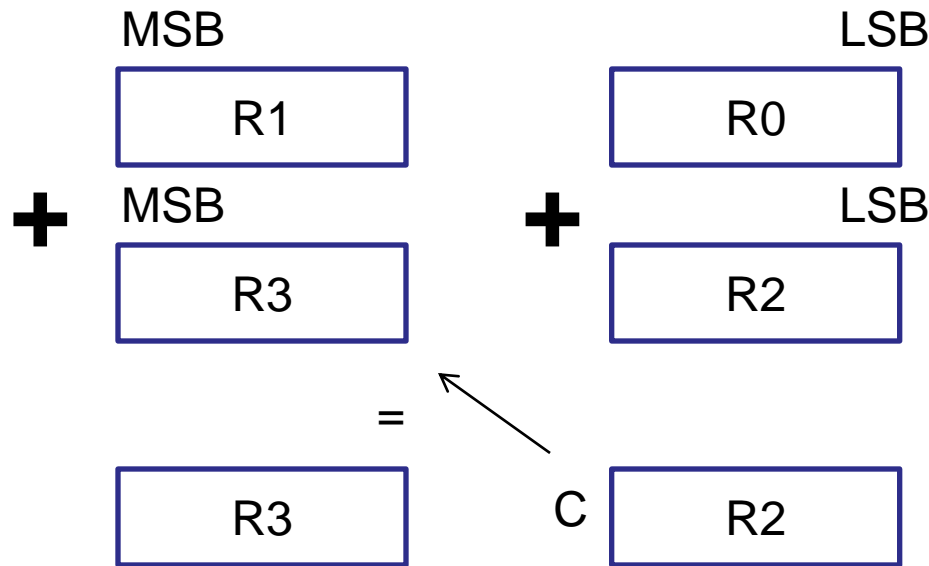


ADD vs. ADDC

- 64-bit add with 32-bit operations

ADDS r2, r2, r0; 32-bit carry out → C

ADC r3, r3, r1 ; .. And added into high word





Example of Condition Code Execution

- Use a sequence of several conditional instructions

```
if (a==0) func(1);  
    CMP     r0,#0  
    MOVEQ   r0,#1  
    BLEQ    func
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;  
if (a>0)  x=1;  
    CMP     r0,#0  
    MOVEQ   r1,#0  
    MOVGT   r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;  
    CMP     r0,#4  
    CMPNE   r0,#10  
    MOVEQ   r1,#0
```



Data processing Instructions

- Consist of :

- Arithmetic:	ADD	ADC	SUB	SBC	RSB	RSC
- Logical:	AND	ORR	EOR	BIC		
- Comparisons:	CMP	CMN	TST	TEQ		
- Data movement:	MOV	MVN				

- These instructions only work on registers, NOT memory.

- Syntax:

`<Operation>{<cond>}{S} Rd, Rn, Operand2`

- Comparisons set flags only - they do not specify Rd
- Data movement does not specify Rn

- Second operand is sent to the ALU via barrel shifter.



Multiply

- Syntax:

- MUL{<cond>}{S} Rd, Rm, Rs $Rd = Rm * Rs$
- MLA{<cond>}{S} Rd,Rm,Rs,Rn $Rd = (Rm * Rs) + Rn$
- [U|S]MULL{<cond>}{S} RdLo, RdHi, Rm, Rs $RdHi,RdLo := Rm*Rs$
- [U|S]MLAL{<cond>}{S} RdLo, RdHi, Rm, Rs $RdHi,RdLo := (Rm*Rs)+RdHi,RdLo$

- Cycle time

- Basic MUL instruction
 - 2-5 cycles on ARM7TDMI
 - 1-3 cycles on StrongARM/XScale
 - 2 cycles on ARM9E/ARM102xE
- +1 cycle for ARM9TDMI (over ARM7TDMI)
- +1 cycle for accumulate (not on 9E though result delay is one cycle longer)
- +1 cycle for “long”

- Above are “general rules” - refer to the TRM for the core you are using for the exact details



Multiply Instructions

Opcode [23:21]	Mnemonic	Meaning	Effect
000	MUL	Multiply (32-bit result)	$Rd := (Rm * Rs)[31:0]$
001	MLA	Multiply-accumulates (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	$RdHi:RdLo += Rm * Rs$

- RdHi:RdLo: 64-bit format RdHi: MSB 32 bits, RdLo: LSB 32 bits
- N: Rd[31] or RdHi[31]
- Z: Rd or RdHi and RdLo are Zero
- C: meaningless
- V: unchanged
- Early ARM supports only 32 bits Multiply operations. 64 bit multiply instructions are supported from ARM7.



Multiplies

- `MUL r4, r3, r2 ; r4 := (r3 x r2) [31:0]`
 - Immediate second operands are not supported
 - Load the value into the register
 - Use shift operations
 - The result register must not be the same as the first source register
 - If the 's' bit is set the V flag is preserved and the 'C' flag is rendered meaningless
- `MLA r4,r3,r2,r1; r4:= (r3 x r2 + r1) [31:0]`



Data Transfer Instructions

- Data transfer between registers and memory.
- Single word and unsigned byte data transfer instructions
- Half-word and signed byte data transfer instructions
- Multiple register transfer instructions
 - Copy subset or multiple registers to memory
- Swap memory and register instructions (SWP)
- Status register to general register transfer instructions



Single register data transfer

LDR	STR	Word
LDRB	STRB	Byte
LDRH	STRH	Halfword
LDRSB		Signed byte load
LDRSH		Signed halfword load

- Memory system must support all access sizes
- Syntax:
 - `LDR{<cond>}{<size>} Rd, <address>`
 - `STR{<cond>}{<size>} Rd, <address>`

e.g. `LDREQB`



Memory Indexing

- Register indirect memory addressing
 - LDR r0, [r1] ; r0 := mem₃₂[r1]
 - STR r0, [r1] ; mem₃₂[r1] := r0
- Particular location:
 - Set base register
 - an address within 4K bytes of the location
- Base plus offset addressing
 - LDR r2, [r1, #4] ; r0 := mem₃₂[r1+4]
 - LDR r2, [r1, #4]! ; r0 := mem₃₂[r1+4]; r1 := r1+4
 - ! Indicates update the base register
- Post-indexed register
 - LDR r2, [r1], #4 ; r0 := mem₃₂[r1]; r1 := r1+4

Address accessed



- Address accessed by LDR/STR is specified by a base register plus an offset
- For word and unsigned byte accesses, offset can be
 - An unsigned 12-bit immediate value (ie 0 - 4095 bytes).

```
LDR r0, [r1, #8]
```

- A register, optionally shifted by an immediate value

```
LDR r0, [r1, r2]
```

```
LDR r0, [r1, r2, LSL#2]
```

- This can be either added or subtracted from the base register:

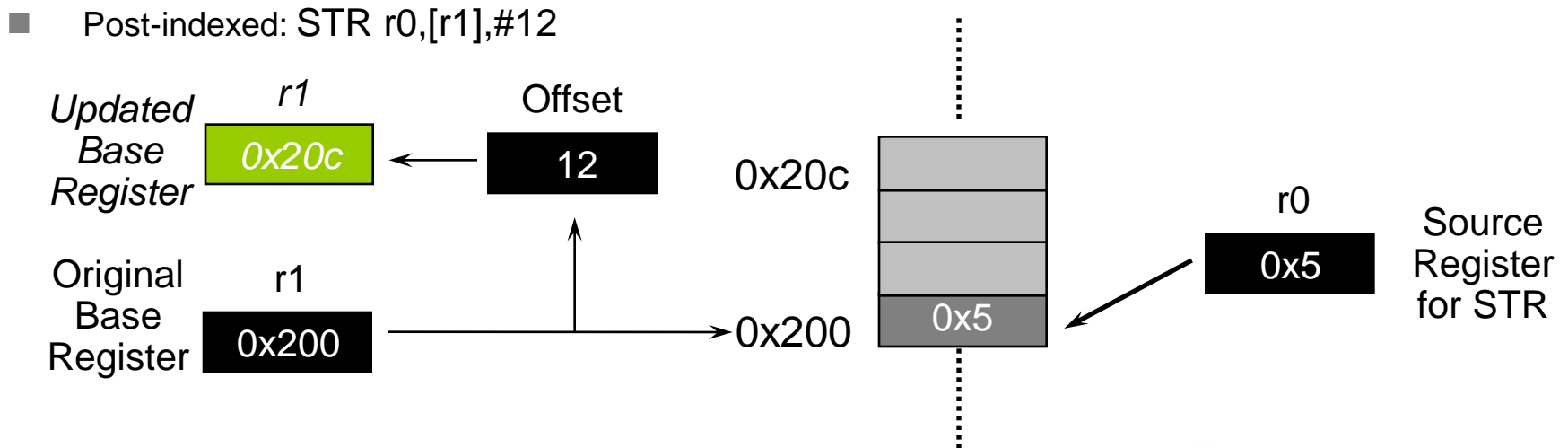
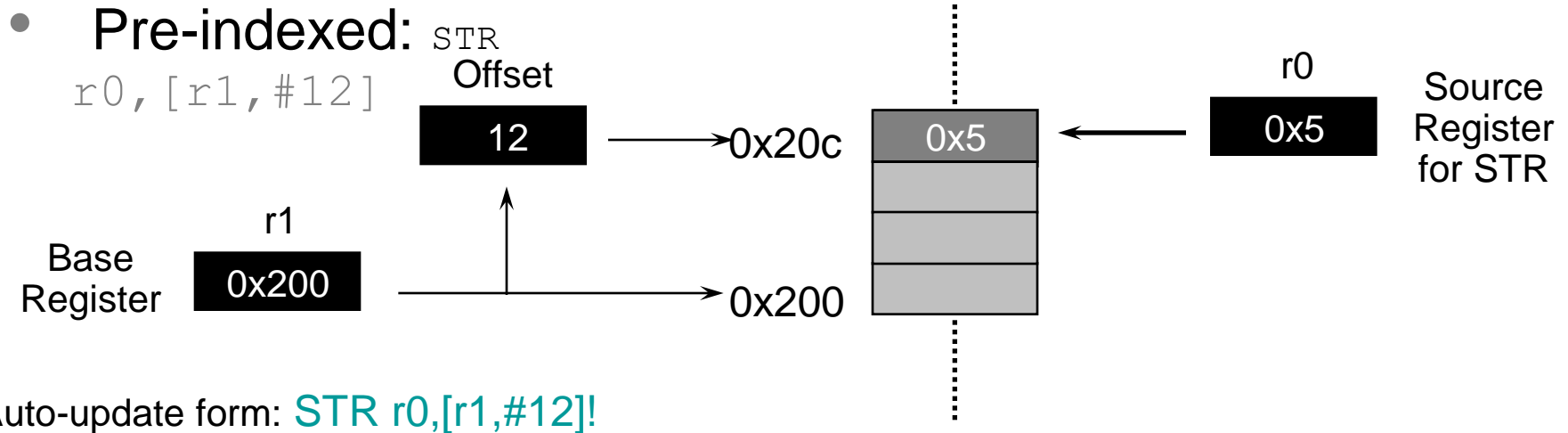
```
LDR r0, [r1, #-8]
```

```
LDR r0, [r1, -r2]
```

```
LDR r0, [r1, -r2, LSL#2]
```

- For halfword and signed halfword / byte, offset can be:
 - An unsigned 8 bit immediate value (ie 0-255 bytes).
 - A register (unshifted).
- Choice of *pre-indexed* or *post-indexed* addressing

Pre or Post Indexed Addressing?



LDM / STM operation



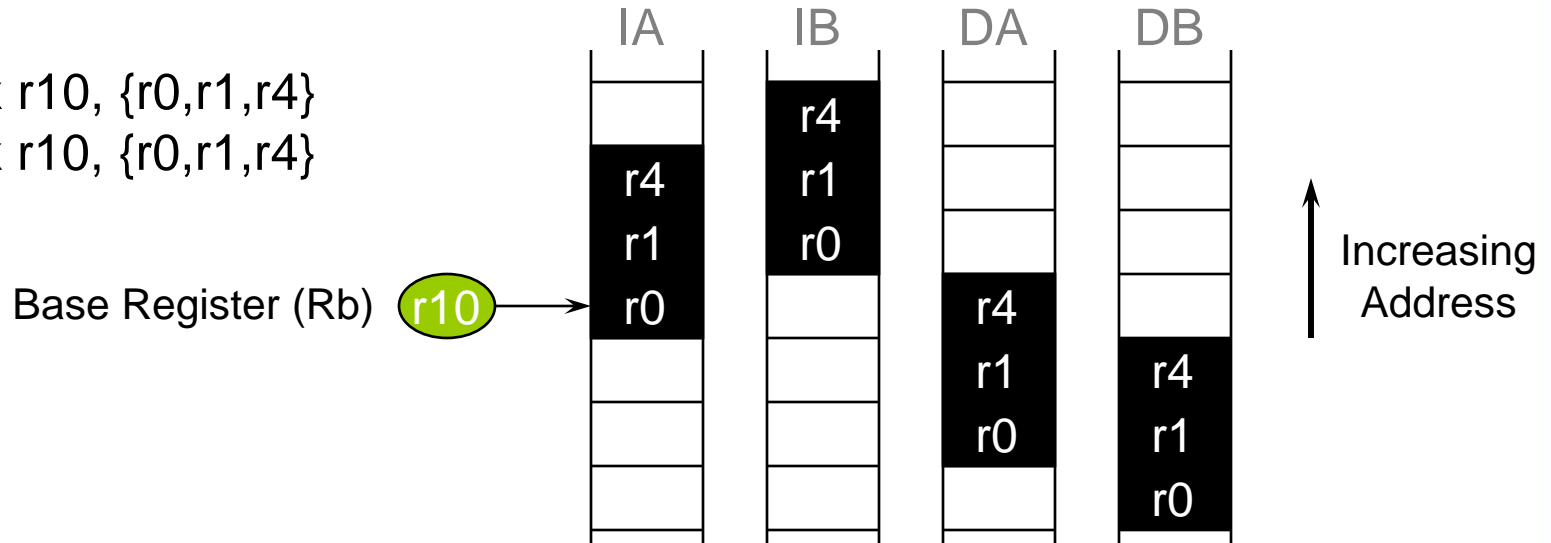
- Syntax:

`<LDM|STM>{<cond>}<addressing_mode> Rb{!}, <register list>`

- 4 addressing modes:

LDMIA / STMIA	increment after
LDMIB / STMIB	increment before
LDMDA / STMDA	decrement after
LDMDB / STMDB	decrement before

LDMxx r10, {r0,r1,r4}
STMxx r10, {r0,r1,r4}



Multiple Register Load and Store Instructions

- Used for transferring large quantities of data
- Usage: procedure entry & exit
- LDMIA r1, {r0, r2, r5}; r0 := mem₃₂[r1]
 ; r2 := mem₃₂[r1+4]
 ; r5 := mem₃₂[r1+8]

r1 should be aligned

If you put r15 in {}, it will change control flow

You can combine with **! also.**



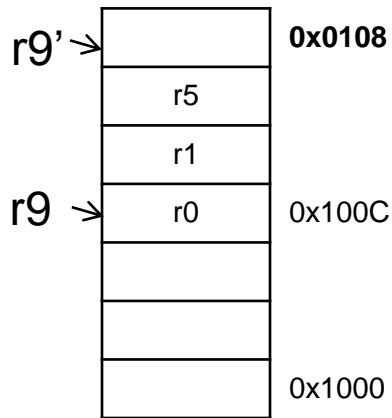
Stack Addressing

- Ascending stack
- Descending stack
- Full stack
- Empty stack

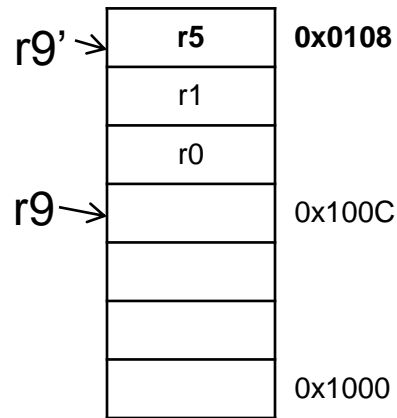


Block Copy Addressing

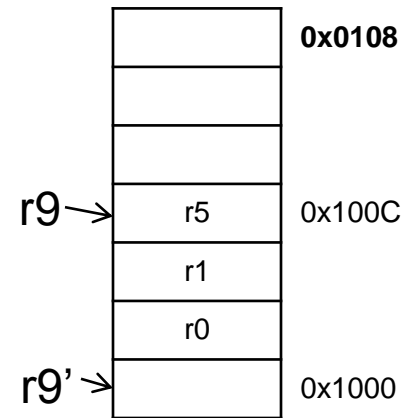
- Auto-indexing, multiple registers
- STMIA, LDMIA
- IA, IB, DA, DB (increment after, increment before, decrement after, decrement before)



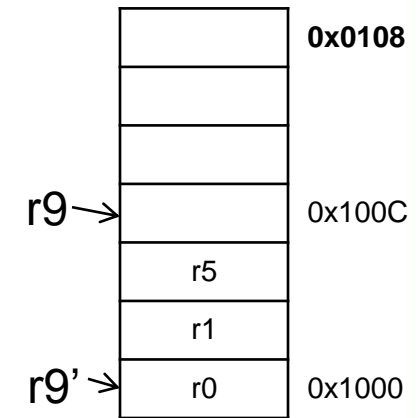
STMIA r9!, {r0, r1, r5}



STMIB r9!, {r0, r1, r5}



STMDA r9!, {r0, r1, r5}



STMDB r9!, {r0, r1, r5}



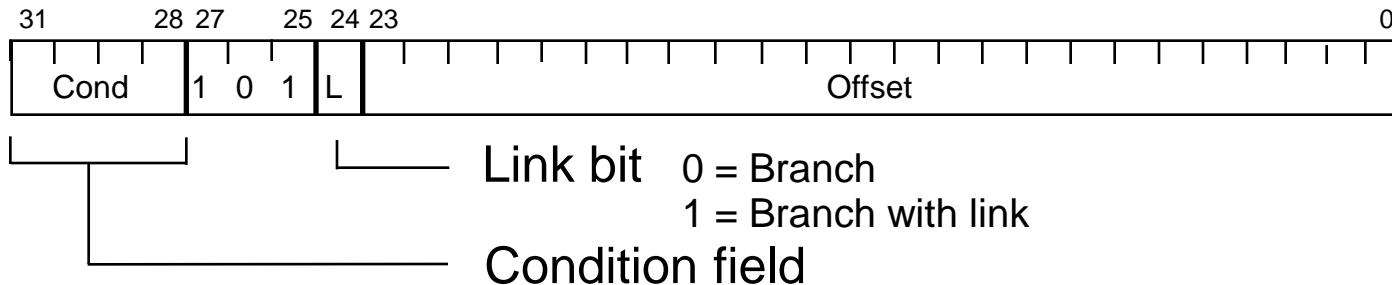
Lab #4

- From 1:10 at States Lab
- Only one demo from each team.



Branch instructions

- Branch : `B{<cond>} label`
- Branch with Link : `BL{<cond>} subroutine_label`



- The processor core shifts the offset field **left by 2 positions**, sign-extends it and adds it to the PC
 - ± 32 Mbyte range
 - How to perform longer branches?



Nested Sub-routine Calls

- Nested sub-routine calls
- Link register (r14) needs to be stored

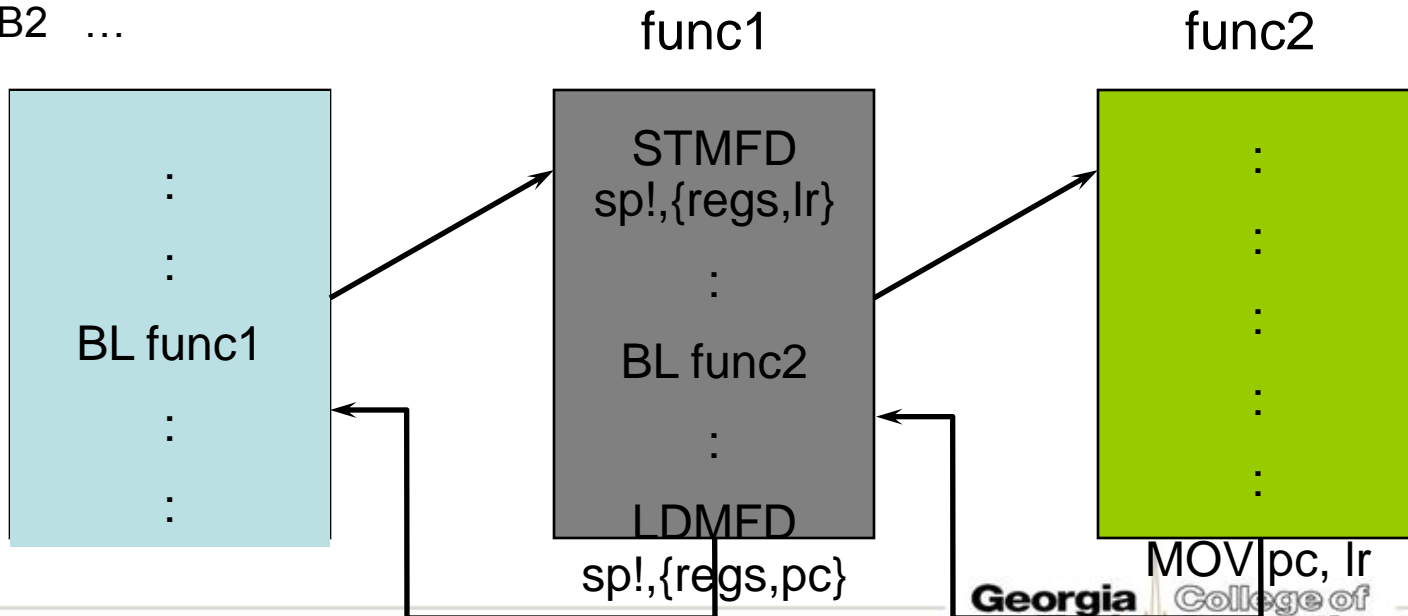
BL SUB1

....

SUB1 STMFD r13!, { r0-r2, r14 } ; save work and link regs

BL SUB2

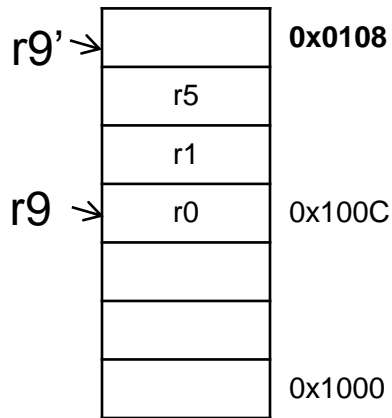
SUB2 ...



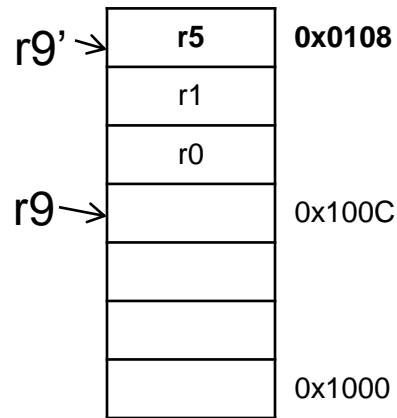


Reviews: Block Copy Addressing

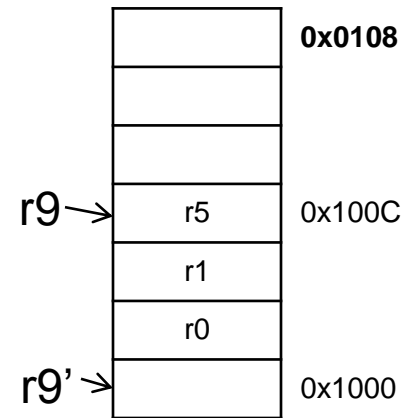
- Auto-indexing, multiple registers
- STMIA, LDMIA
- IA, IB, DA, DB (increment after, increment before, decrement after, decrement before)



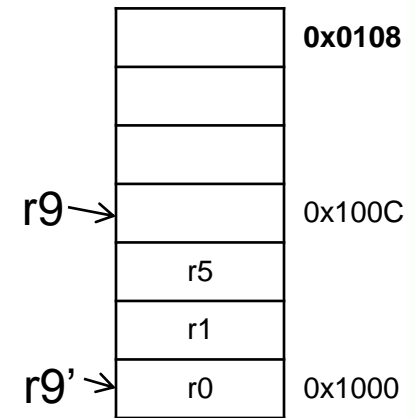
STMIA r9!, {r0, r1, r5}



STMIB r9!, {r0, r1, r5}



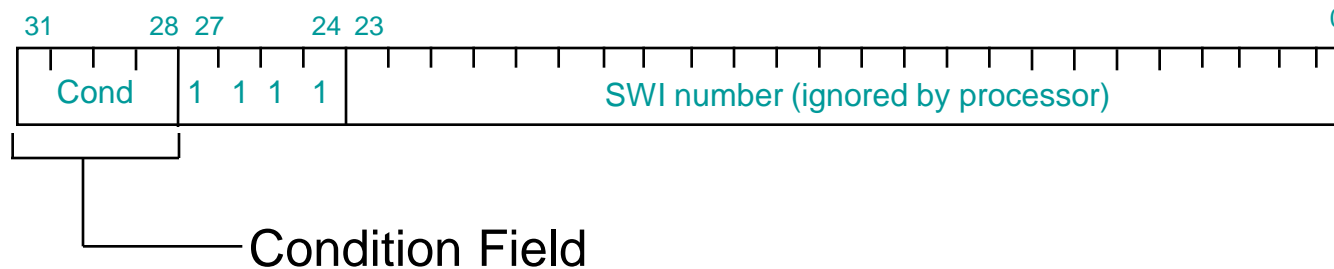
STMDA r9!, {r0, r1, r5}



STMDB r9!, {r0, r1, r5}

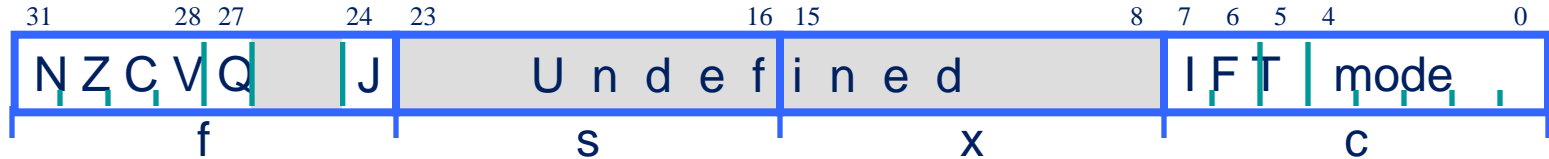


Software Interrupt (SWI)



- Causes an exception trap to the SWI hardware vector
- The SWI handler can examine the SWI number to decide what operation has been requested.
- By using the SWI mechanism, an operating system can implement a set of privileged operations which applications running in user mode can request.
- Syntax:
 - `SWI{<cond>} <SWI number>`

PSR Transfer Instructions



- MRS and MSR allow contents of CPSR / SPSR to be transferred to / from a general purpose register.
- Syntax:

— `MRS{<cond>} Rd,<psr>` ; `Rd = <psr>`

— `MSR{<cond>} <psr[_fields]>,Rm` ; `<psr[_fields]> = Rm`

where

- `<psr>` = CPSR or SPSR
 - `[_fields]` = any combination of 'fsxc'
 - Also an immediate form
- `MSR{<cond>} <psr_fields>,#Immediate`
- In User Mode, all bits can be read but only the condition flags (`_f`) can be written.



Exceptions/Interrupts

- 1. Exceptions generated as the direct effect of executing an instruction
 - Software interrupt, undefined instructions, prefetch aborts
- 2. Exceptions generated as a side-effect of an instruction
 - Memory fault during a load or store data access
 - Unaligned access
- 3. Exceptions generated externally, unrelated to the instruction flow. Reset, IRQ, and FIQ



Exception/Interrupt

- Interrupt:
 - It handles as soon as the current instruction is finished
 - E.g.) External events, Fast interrupt (FIQ)
- Exception
 - It handles immediately
 - E.g.) page faults, unaligned accesses, undefined opcode



Sequences of Actions

- 1) changes to the operation mode corresponding to the particular exception
- 2) saves the next PC address into the corresponding r14 register.
- 3) Saves the old value of CPSR in the SPSR of the new mode
- 4) Disables IRQs by setting bit 7 of the CPSR
 - For a fast interrupt, disables further fast interrupt by setting bit 6 of the CPSR. (no nested fast interrupts!)
- 5) Set PC address to the corresponding interrupt vector table



Exception Vector Address

Exception	Mode	Vector Address	Priority
Reset	SVC	0x00000000	1
Undefined instruction	UND	0x00000004	6
Software interrupt (SWI)	SVC	0x00000008	6
Prefetch abort (instruction fetch memory fault)	Abort	0x0000000C	5
Data abort (data access memory fault)	Abort	0x00000010	2
IRQ (normal interrupt)	IRQ	0x00000018	4
FIQ (Fast interrupt)	FIQ	0x0000001C	3

- Vector address contains a branch to the relevant routine, except FIQ
 - No space to put code.
- FIQ code can start immediately because it has the highest vector address.



Registers

- Two banked registers to hold the return address and a stack pointer
- Stacks are used to store registers
 - Callee based register saving
- FIQ → additional registers
 - Why? To save time to save registers



Exception Return

- After the exception handler, the hardware just starts from the user mode.
- **Software** must
 - Restore the modified registers
 - CPSR must be restored from the appropriate SPSR
 - PC must be changed back to the relevant instruction address in the user instruction stream
 - These two cannot happen independently



Atomic Operations

- Return using a link register (r14)

MOV^S pc, r14

S bit is set, a branch occurs and the SPSR of the current mode is copied to the CPSR

- Return using a stack

LDFMD sp!, {r0-r12, pc}[^]

- The ^ qualifier specifies that the CPSR is restored from the SPSR. It must be used only from a privileged mode.



Return Address

- IRQ and FIQ must return **one instruction** early in order to execute the instruction that raised an exception
- Prefetch abort must return **one instruction** early to execute the instruction that had caused a memory fault when first requested
- Data abort must return the instruction that caused exception.



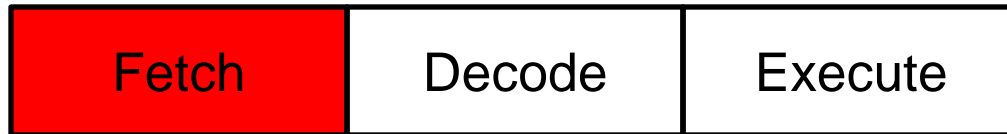
Use of R15

- R15: PC
 - PC may be used as a source operand
 - Register-based shift cannot use R15 as source operands.
- **Running-ahead** PC's behavior
 - PC is always running ahead
 - PC is always pointing +8 of the current instruction
 - Imagine 3-stage pipeline machine . PC is pointing what to fetch when an instruction is in the WB stage in the 3-stage pipeline machine
- When R15 is a source, the current PC + 8 is supplied to the source operand.
- When R15 is a destination
 - S: 1: SPSR → CPSR, affecting interrupt, resource PC and CPSR automatically,



Exception generation time

- Pre-fetch abort : instruction fetch



PC+4

PC+8

- Data abort : memory execution



Control Flow Instructions

	Return Instruction	Previous State	
		ARM R14_x	THUMB R14_x
BL	MOV PC, R14	PC + 4	PC + 2
SWI	MOVS PC, R14_svc	PC + 4	PC + 2
UDEF	MOVS PC, R14_und	PC + 4	PC + 2
FIQ	SUBS PC, R14_fiq, #4	PC + 4	PC + 4
IRQ	SUBS PC, R14_irq, #4	PC + 4	PC + 4
PABT	SUBS PC, R14_abt, #4	PC + 4	PC + 4
DABT	SUBS PC, R14_abt, #8	PC + 8	PC + 8
RESET	NA	-	-



Controlling Interrupts

```
void event_EnableIRQ (void)
{
  __asm {
    MRS r1, CPSR
    BIC r1, r1, #0x80
    MSR CPSR_c, r1
  }
}
```

Enable Bit 7 (set register 0)

```
void event_DisableIRQ (void)
{
  __asm {
    MRS r1, CPSR
    ORR r1, r1, #0x80
    MSR CPSR_c, r1
  }
}
```

Disable bit 7 (set 1)



Bit 7: interrupt

Bit 6: Fast interrupt



Typical example of interrupt handler

- SUB lr, lr, #4
 - STMFD sp!{reglist, lr}
- ;
- LDMFD sp!, {reglist,pc}^



Announcement

- All grades should be at T-square including Lab #4
- Lab #6 demo during the class on Friday