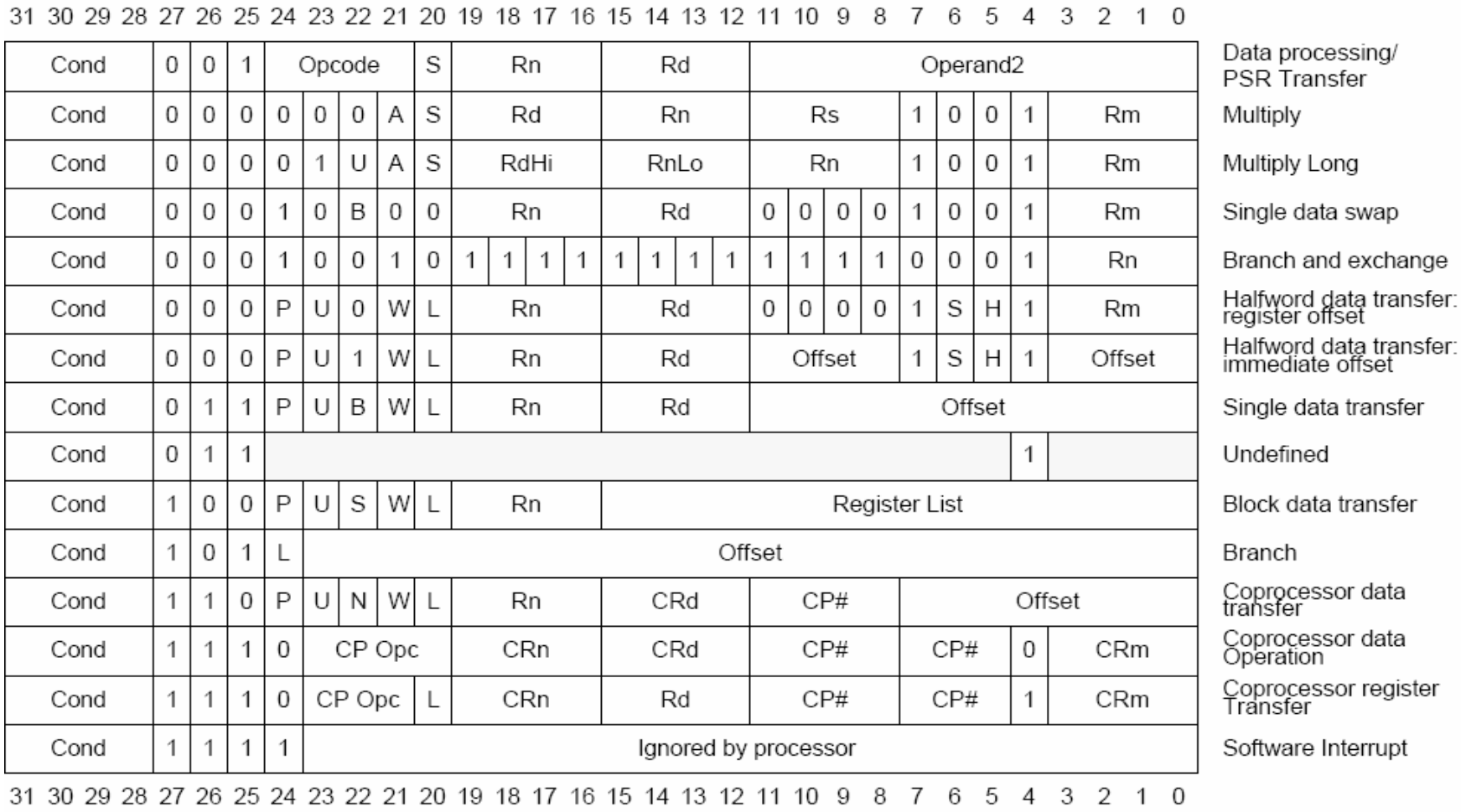
Outline

- Data Processing Instructions
- Branch Instructions
- Load-store instructions
- Software interrupt instructions
- Program status register instructions
- Conditional Execution

ARM Instruction Set Format



6.1 Data Processing Instructions

- Manipulate data *within* registers
- Data processing instructions
  - Move instructions
  - Arithmetic instructions
  - Logical instructions
  - Comparison instructions
  - Multiply instructions

6.1.1 Move Instruction

- Syntax: <instruction> {<cond>}{S}Rd, N
  - N: a register or immediate value

- MOV : move
  - MOV r0, r1; r0 = r1
  - MOV r0, #5; r0 = 5
- MVN : move (negated)
  - MVN r0, r1; r0 = NOT(r1)=~(r1)

Preprocessed by Shifter

- ***Example 1***
  - PRE: r5 = 5, r7 = 8;
  - MOV r7, r5, LSL #2; r7 = r5 << 2 = r5*4
  - POST: r5 = 5, r7 = 20

6.1.2 Preprocessed by Shifter

- LSL: logical shift left
  - x << y, the least significant bits are filled with zeroes
- LSR: logical shift right:
  - (unsigned) x >> y, the most significant bits are filled with zeroes
- ASR: arithmetic shift right
  - (signed) x >> y, copy the sign bit to the most significant bit
- ROR: rotate right
  - ((unsigned) x >> y) | (x << (32-y))
- RRX: rotate right extended
  - c flag <<31 | ((unsigned) x >> 1)
  - Performs 33-bit rotate, with the CPSR's C bit being inserted above sign bit of the word

Preprocessed by Shifter (Cont.)

- ***Example 2***
  - PRE: r0 = 0x00000000, r1 = 0x80000004
  - MOV r0, r1, LSL #1 ; r0 = r1 *2
  - POST r0 = **0x00000008**, r1 = 0x80000004

6.1.3 Arithmetic Instructions

- Syntax: <instruction> {<cond>} {S} Rd, Rn, N
  - N: a register or immediate value
- ADD : add
  - ADD r0, r1, r2; r0 = r1 + r2
- ADC : add with carry
  - ADC r0, r1, r2; r0 = r1 + r2 + C
- SUB : subtract
  - SUB r0, r1, r2; r0 = r1 - r2
- SBC : subtract with carry
  - SUC r0, r1, r2; r0 = r1 - r2 + C -1

6.1.3 Arithmetic Instructions (Cont.)

- RSB : reverse subtract
  - RSB r0, r1, r2; r0 = r2 – r1
- RSC : reverse subtract with carry
  - RSC r0, r1, r2; r0 = r2 – r1 + C -1
- MUL : multiply
  - MUL r0, r1, r2; r0 = r1 x r2
- MLA : multiply and accumulate
  - MLA r0, r1, r2, r3; r0 = r1 x r2 + r3

6.1.4 Logical Operations

- Syntax: <instruction>{<cond>}{S}Rd, RN, N
  - N: a register or immediate value
- AND : Bit-wise and
- ORR : Bit-wise or
- EOR : Bit-wise exclusive-or
- BIC : bit clear
  - BIC r0, r1, r2; r0 = r1 & Not(r2)

Logical Operations (Cont)

- ***Example 3:***
  - PRE: r1 = 0b1111, r2 = 0b0101
  - BIC r0, r1, r2 ; r0 = r1 AND (NOT(r2))
  - POST: r0=0b1010

6.1.5 Comparison Instructions

- Compare or test a register with a 32-bit value
  - Do not modify the registers being compared or tested
  - But only set the values of the *NZCV* bits of the *CPSR register*
    - Do not need to apply to **S** suffix for comparison instruction to update the flags in CPSR register

Comparison Instructions (Cont.)

- Syntax: <instruction> {<cond>} {S} Rd, N
  - N: a register or immediate value
- CMP : compare
  - CMP r0, r1; compute (r0 - r1) and set NZCV
- CMN : negated compare
  - CMP r0, r1; compute (r0 + r1) and set NZCV
- TST : bit-wise AND test
  - TST r0, r1; compute (r0 AND r1) and set NZCV
- TEQ : bit-wise exclusive-or test
  - TEQ r0, r1; compute (r0 EOR r1) and set NZCV

Comparison Instructions (Cont.)

- ***Example 4***
  - PRE: CPSR = nzcvqiFt_USER, r0 = 4, r9 = 4
  - `CMP r0, r9`
  - POST: CPSR = nZcvqiFt_USER

6.1.6 Multiply Instruction

- Syntax:
  - MLA{<cond>}{S}Rd, Rm, Rs, Rn
  - MUL{<cond>}{S}Rd, Rm, Rs
- MUL : multiply
  - MUL r0, r1, r2; r0 = r1*r2
- MLA : multiply and accumulate
  - MLA r0, r1, r2, r3; r0 = (r1*r2) + r3

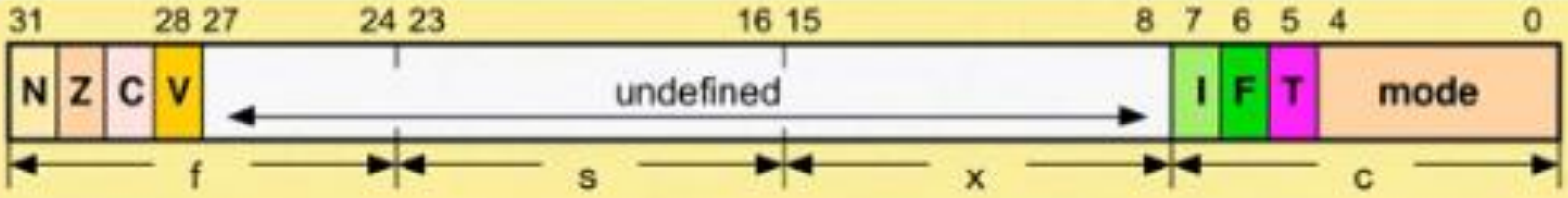
Multiply Instruction (Cont.)

- Syntax: <instruction>{<cond>} {S} RdLo, RdHi, Rm, Rs
  - Multiply onto *a pair of register* representing a 64-bit value
- UMULL : unsigned multiply long
  - UMULL r0, r1, r2, r3; [r1,r0] = r2*r3
- UMLAL : unsigned multiply accumulate long
  - UMLAL r0, r1, r2, r3; [r1,r0] = [r1,r0]+(r2*r3)
- SMULL: signed multiply long
  - SMULL r0, r1, r2, r3; [r1,r0] = r2*r3
- SMLAL : signed multiply accumulate long
  - SMLAL r0, r1, r2, r3; [r1,r0] = [r1,r0]+(r2*r3)

6.2 Branch Instructions

- Branch instruction
  - Change the flow of execution
  - Used to call a routine
- Allow applications to
  - Have *subroutines*
  - Implement *if-then-else* structure
  - Implement *loop* structure

CPSR



Condition code flags

- N = Negative result from ALU
- Z = Zero result from ALU
- C = ALU operation Carried out
- V = ALU operation oVerflowed

Mode bits

10000	User
10001	FIQ
10010	IRQ
10011	Supervisor
10111	Abort
11011	Undefined
11111	System

Interrupt Disable bits.

I = 1: Disables the IRQ.

F = 1: Disables the FIQ.

T Bit (Arch. with Thumb mode only)

T = 0: Processor in ARM state

T = 1: Processor in Thumb state

LR


User, SYS
FIQ
IRQ
SVC
Undef
Abort
r0
r1
r2
r3
r4
r5
r6
r7
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
r15 (pc)
cpsr
User mode r0-r7, r15, and cpsr
r8
r9
r10
r11
r12
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr
User mode r0-r12, r15, and cpsr
r13 (sp)
r14 (lr)
spsr

Control Flow Instructions

- These instructions change the order of instruction execution or to jump from one memory location to other.
- Types of conditional flow instructions:
  - Unconditional branch
  - Conditional branch
  - Branch and Link

Branching Instructions

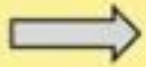
- Unconditional branch : unconditionally loads the PC with the specified address.

```
            B       Target
            ...
            ...
Target      ...
```

- Conditional branch: Conditionally loads the PC with the specified address.

• An example: `if (r2 != 10) r5 = r5 +10 - r3`

```
        CMP     r2,#10
        BEQ     SKIP
        ADD     r5,r5,r2
        SUB     r5,r5,r3
SKIP    ...
```



```
CMP         r2,#10
ADDNE       r5,r5,r2
SUBNE       r5,r5,r3
```

Branching Instructions

- In ARM there are four important branch instructions available,

- **B** **Branch**
- **BL** **Branch with link**
- **BX** **Branch Exchange**
- **BLX** **Branch Exchange with link**

Branching Instructions - B and BL

- Instruction Format:
  - Branch: **B{<cond>} Label**
  - Branch with Link: **BL{<cond>} subroutine_label**
- **B** – Branch

  PC = <address> or <label>
- **BL** – Branch with Link

  R14 = address of next instruction, PC = <address>
- Thus to return from a linked branch
- MOV r15,r14 or

  MOV pc,lr

Conditional Branch Instructions

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

Examples - Branching Instructions

- **B Instruction :**

```
Back : MOV r0,r1
       ADD r0,r1,r2
       SUB r3,r0,r1
       B Back
```

```
    ...             ; some code here
    B fwd           ; jump to label 'fwd'
    ...             ; more code here
fwd
```

- **BL Instruction**

```
       CMP r2,#10
       BL Next
       ADDEQ r2,r3,r4
Next:   AND r4,r5,r6
       MOV PC,lr
```

```
    ...
    ...
    BL  calc        ; call 'calc'
    ...             ; returns to here
    ...

calc                ; function body
    ADD r0, r1, r2  ; do some work here
    MOV pc, r14     ; PC = R14 to retu
```

Examples - Branching Instructions

- **B Instruction :**

```
        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 Instructions (Cont.)

- Syntax
  - B{<cond>}lable
  - BL{<cond>}lable
- B : branch
  - B label; pc (program counter) = label
  - Used to change execution flow
- BL : branch and link
  - BL label; pc = label, lr = address of the next address after the BL
  - Similar to the B instruction but can be used for subroutine call
    - Overwrite the link register (lr) with a return address

Branch Instructions (Cont.)

- ***Example 5***

```
            B forward
            ADD   r1,   r2,
            #4    ADD   r0,
            r6,   #2    ADD
            r3, r7, #4
Forward
            SUB r1, r2, #4
Backward
            SUB r1, r2, #4
            B backward
```

Branch Instructions (Cont.)

- ***Example 6:***

```
            BL subroutine
            CMP r1, #5
            MOVEQ r1, #0
            …
subroutine
            <subroutine code>
            MOV pc, lr ;  return by moving pc = lr
```

6.3 Load-Store Instructions

- Transfer data between memory and processor registers

- Three types
  - *Single-register transfer*
  - *Multiple-register transfer*
  - *Swap*

6.3.1 Simple-Register Transfer

- Moving a single data item in and out of register

- Data item can be
  - A word (32-bits)
  - Halfword (16-bits)
  - Bytes (8-bits)

Simple-Register Transfer (Cont.)

- Syntax
  - <LDR|STR>{<cond>}{B}Rd,addressing[1]
  - LDR{<cond>}SB|H|SH Rd,addressing[2]
  - STR{<cond>}H Rd,addressing[2]
- LDR : load word into a register from memory
- LDRB : load byte
- LDRSB : load signed byte
- LDRH : load half-word
- LSRSH : load signed halfword
- STR: store word from a register to memory
- STRB : store byte
- STRH : store half-word

Simple-Register Transfer (Cont.)

- ***Example 7***

```
LDR r0, [r1]        ;= LDR r0, [r1, #0]
                    ;r0 = mem32[r1]
STR r0, [r1]        ;= STR r0, [r1, #0]
                    ;mem32[r1]= r0
```

  - Register *r1* is called the *base address register*

6.3.2 Single-Register Load-Store Addressing Mode

- ***Index method***, also called ***Base-Plus-Offset Addressing***
  - Base register
    - r0 – r15
  - Offset, add or subtract an unsigned number
    - Immediate
    - Register (not PC)
    - Scaled register

Single-Register Load-Store Addressing Mode (Cont.)

- ***Preindex***:
  - data: *mem[base+offset]*
  - Base address register: *not updated*
  - Ex: LDR r0,[r1,#4] ; r0:=mem32[r1+4]
- ***Postindex***:
  - data: *mem[base]*
  - Base address register: *base + offset*
  - Ex: LDR r0,[r1],#4 ; r0:=mem32[r1], then r1:=r1+4
- ***Preindex with writeback*** (also called ***auto-indexing***)
  - Data: *mem[base+offset]*
  - Base address register: *base + offset*
  - Ex: LDR r0, [r1,#4]! ; r0:=mem32[r1+4], then r1:=r1+4

Single-Register Load-Store Addressing Mode (Cont.)

- ***Example 8***
  - r0 = 0x00000000, r1 = 0x00009000, mem32[0x00009000] = 0x01010101, mem32[0x00009004] = 0x02020202
  - ***Preindexing:*** LDR r0, [r1, #4]
    - r0 = 0x02020202, r1=0x00009000
  - ***Postindexing:*** LDR r0, [r1], #4
    - r0 = 0x01010101, r1=0x00009004
  - ***Preindexing with writeback:*** LDR r0, [r1, #4]!
    - R0 =0x02020202, r1=0x00009004

Single-Register Load-Store Addressing Mode (Cont.)

Addressing mode and index method	**Addressing syntax**
Preindex with *immediate offset*	[Rn, #+/-offset_12]
Preindex with *register offset*	[Rn, +/-Rm]
Preindex with *scaled register offset*	[Rn, +/-Rm, shift #shift_imm]
Preindex writeback with immediate offset	[Rn, #+/-offset_12]!
Preindex writeback with register offset	[Rn, +/-Rm]!
Preindex writeback with *scaled register offset*	[Rn, +/-Rm, shift #shift_imm]
Immediate postindexed	[Rn], #+/-offset_12]
Register postindexed	[Rn], +/-Rm!
Scaled register postindexed	[Rn], +/-Rm, shift #shift_imm

Examples of LDR Using Different Addressing Modes

	Instruction	r0=	r1+=
Preindex with writeback	LDR r0, [r1, #0x4]!	mem32[r1+0x4]	0x4
	LDR r0, [r1,r2]!	mem32[r1+r2]	r2
	LDR r0,[r1, r2, LSR#0x4]!	mem32[r1+(r2 LSR 0x4)]	(r2 LSR 0x4)
Preindex	LDR r0, [r1, #0x4]	mem32[r1+0x4]	*not updated*
	LDR r0, [r1, r2]	mem32[r1+r2]	*not updated*
	LDR r0, [r1, -r2, LSR #0x4]	Mem32[r1-(r2 LSR 0x4)]	*not updated*
Postindex	LDR r0, [r1], #0x4	mem32[r1]	0x4
	LDR r0, [r1], r2	Mem32[r1]	r2
	LDR r0, [r1], r2 LSR #0x4	mem32[r1]	(r2 LSR 0x4)

6.3.3 Multiple-Register Transfer

- Transfer multiple registers between memory and the processor in a single instruction

- More efficient than single-register transfer
  - Moving blocks of data around memory
  - Saving and restoring context and stack

Multiple Register Load Store or Data Transfer

- Transfer multiple registers between memory and the processor in a single instruction
- LDM load multiple registers
- STM store multiple registers

suffix	**meaning**
IA	increase after
IB	increase before
DA	decrease after
DB	decrease before

e.g. LDMIA, LDMIB

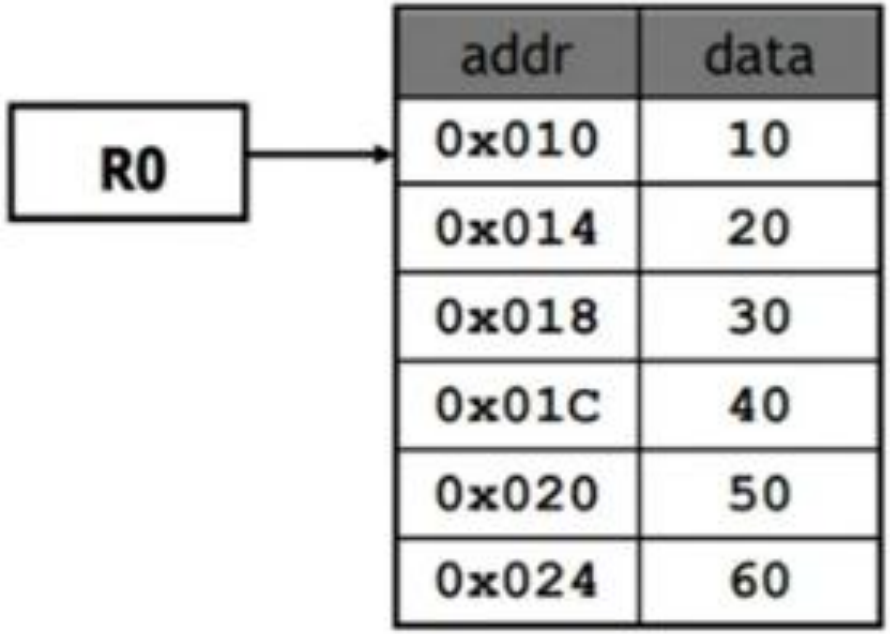
Multiple Register Load Store or Data Transfer

```
LDMIA R0, {R1,R2,R3}
```

or

```
LDMIA R0, {R1-R3}
```

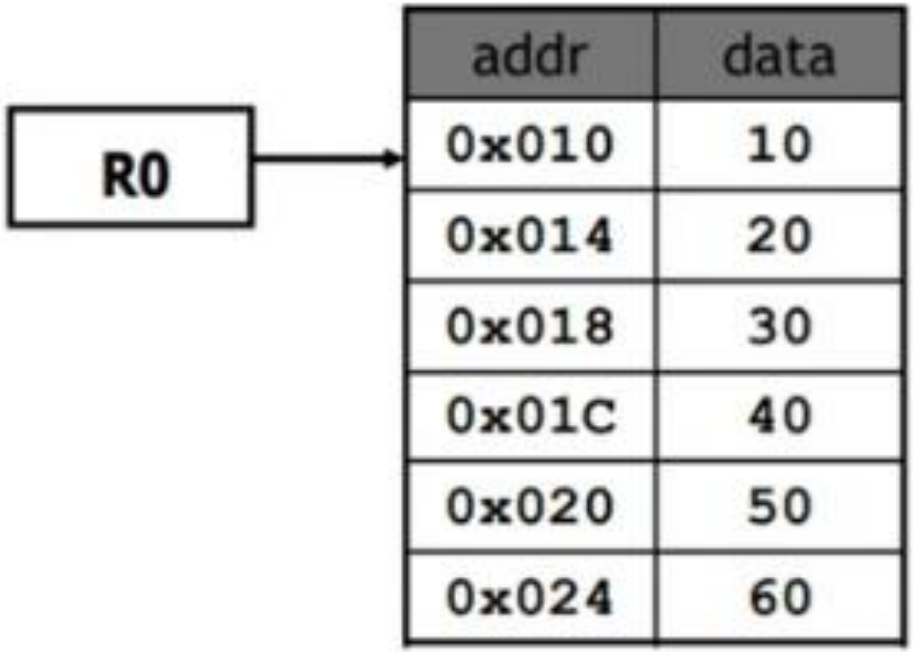
```
R1: 10
R2: 20
R3: 30
R0: 0x10
```



Multiple Register Load Store or Data Transfer

```
LDMIA R0!, {R1,R2,R3}
```

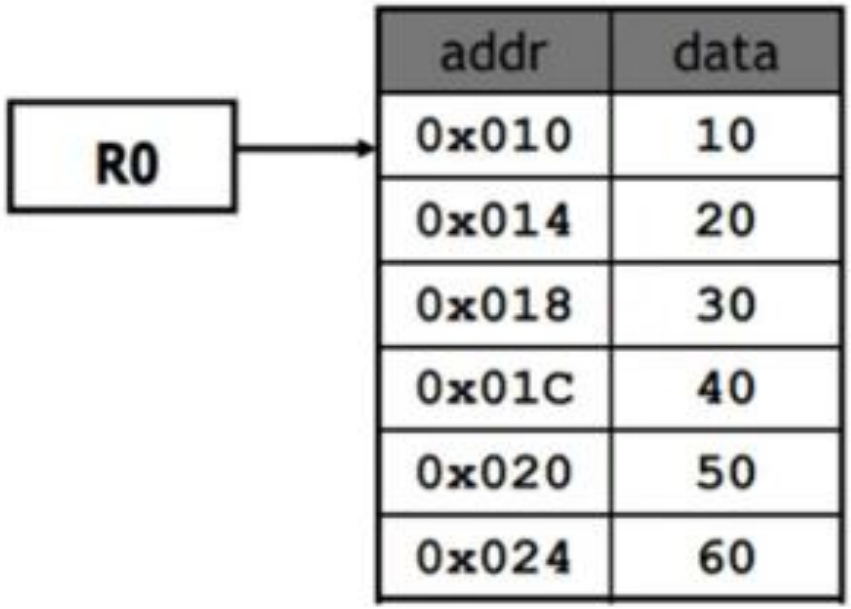
```
R1: 10
R2: 20
R3: 30
R0: 0x01C
```



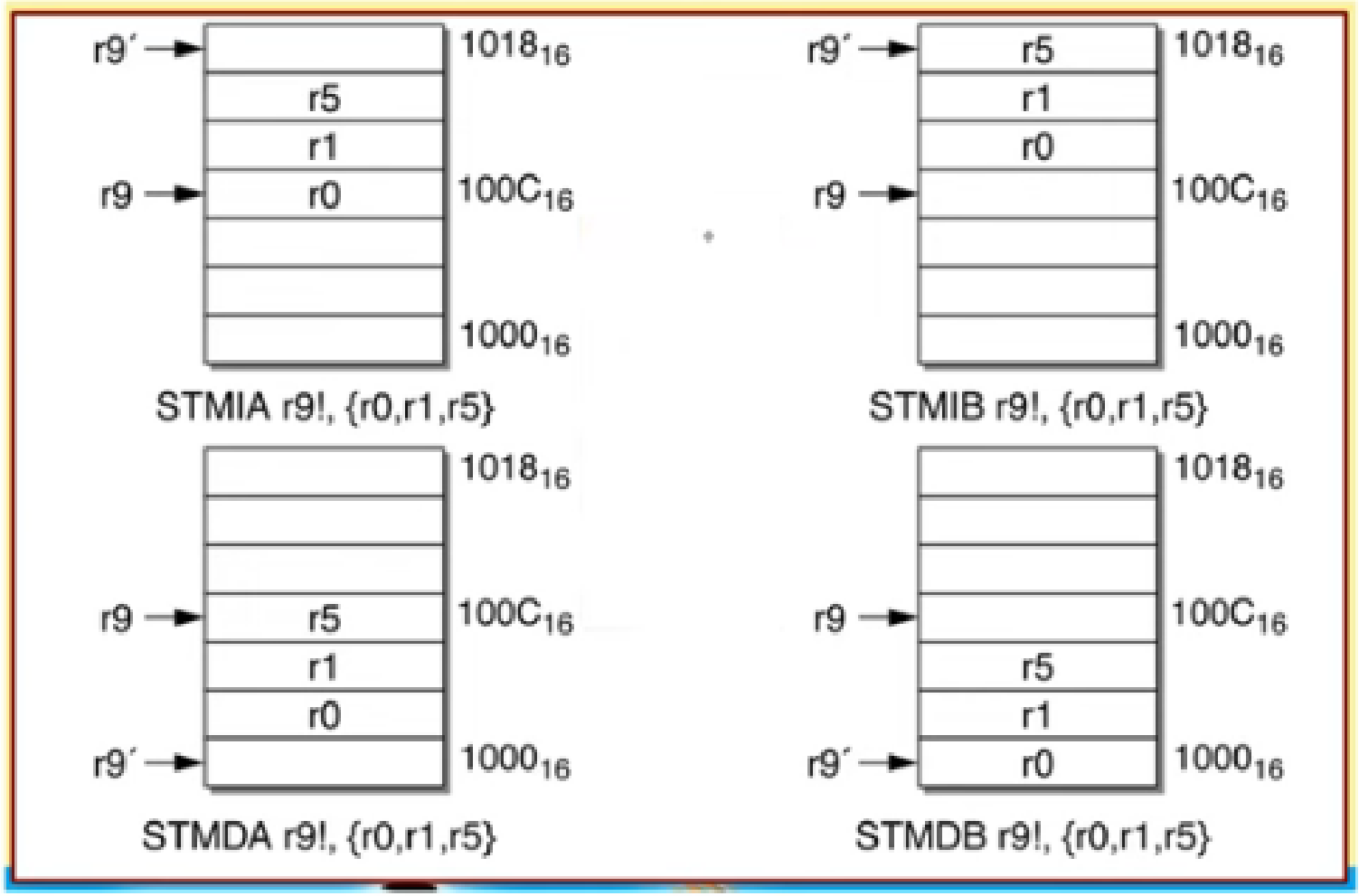
Multiple Register Load Store or Data Transfer

```
LDMIB R0!, {R1,R2,R3}
```

```
R1: 20
R2: 30
R3: 40
R0: 0x01C
```



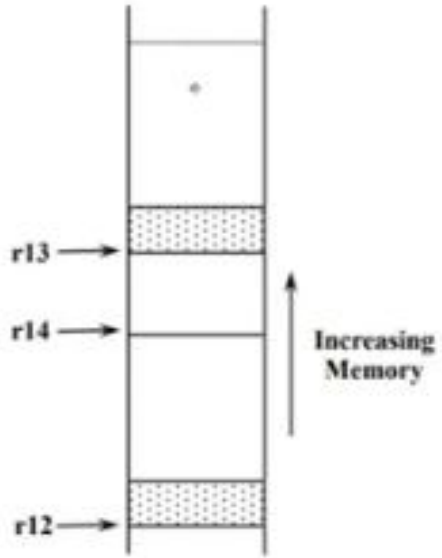
Multiple Register Load Store or Data Transfer



Example: Block Copy

- Copy a block of memory, which is an exact multiple of 12 words long from the location pointed to by r12 to the location pointed to by r13. r14 points to the end of block to be copied.

```
loop :LDMIA r12!, {r0-r11}        ; load 48 bytes
      STMIA r13!, {r0-r11}        ; and store them
      CMP r12, r14                ; check for the end
      BNE loop                    ; and loop until done
```



- This loop transfers 48 bytes in 31 cycles Over 50 Mbytes/sec at 33 MHz

Multiple-Register Transfer (Cont.)

- Load-store multiple instruction can increase interrupt latency
  - Interrupt can be occurred after an instruction has been completed
  - Each load multiple instruction takes $2 + N*t$ cycles
    - N: the number of registers to load
    - t: the number of cycles required for sequential access to memory
  - Compilers provides a switch to control the maximum number of registers between transferred
    - Limit the maximum interrupt latency

Multiple-Register Transfer (Cont.)

- Syntax:
  - <LDM|STM>{<cond>} <mode> *Rn*{!}, <registers>{^}
  - Address mode: See the next page
  - ^: optional
    - Can not be used in User Mode and System Mode
    - If *op* is LDM and *reglist* contains the pc (r15)
      - SPSR is also copied into the CPSR.
    - Otherwise, data is transferred into or out of the User mode registers instead of the current mode registers.

Addressing Mode

Addressing mode	**Description**	**Start address**	**End address**	**Rn!**
IA	increment address after each transfer	*Rn*	*Rn*+4**N* -4*	*Rn*+4**N*
IB	increment address before each transfer	*Rn* + 4	*Rn*+4**N*	*Rn*+4**N*
DA	decrement address after each transfer	*Rn*-4**N* +4*	*Rn*	*Rn*-4**N*
DB	decrement address before each transfer	*Rn*-4**N*	*Rn* – 4	*Rn*+4**N*

Multiple-Register Transfer (Cont.)

- ***Example 9***
  - **PRE:**

    mem32[0x80018] = 0x03,

    mem32[0x80014] = 0x02,

    mem32[0x80010] = 0x01,

    r0 = 0x00080010,

    r1 = r2 = r3= 0x00000000

  - LDMIA r0!, {r1-r3}, or LDMIA r0!, {r1, r2, r3}
    - Register can be explicitly listed or use the “-” character

Pre-Condition for LDMIA Instruction

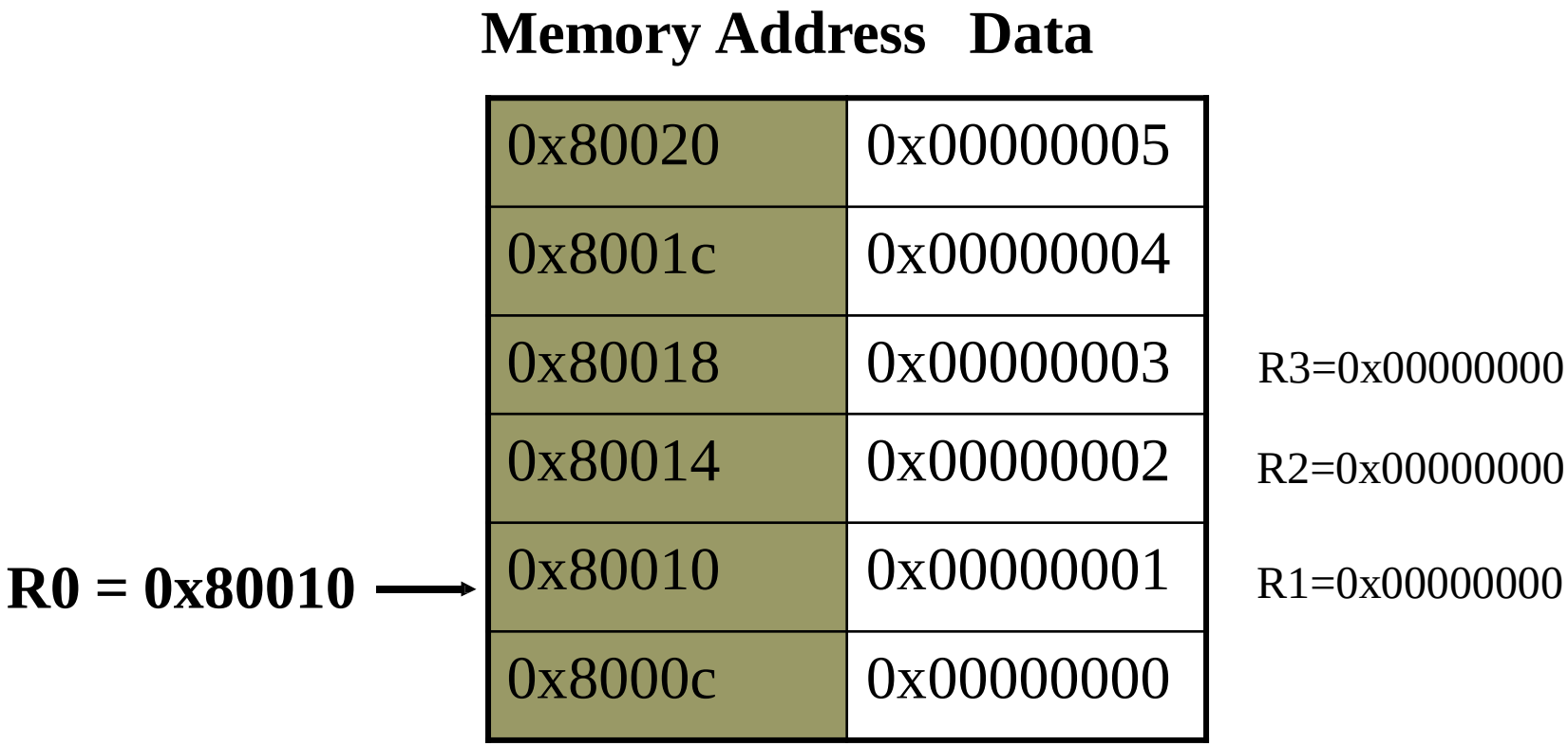


Figure 1

Post-Condition for LDMIA Instruction

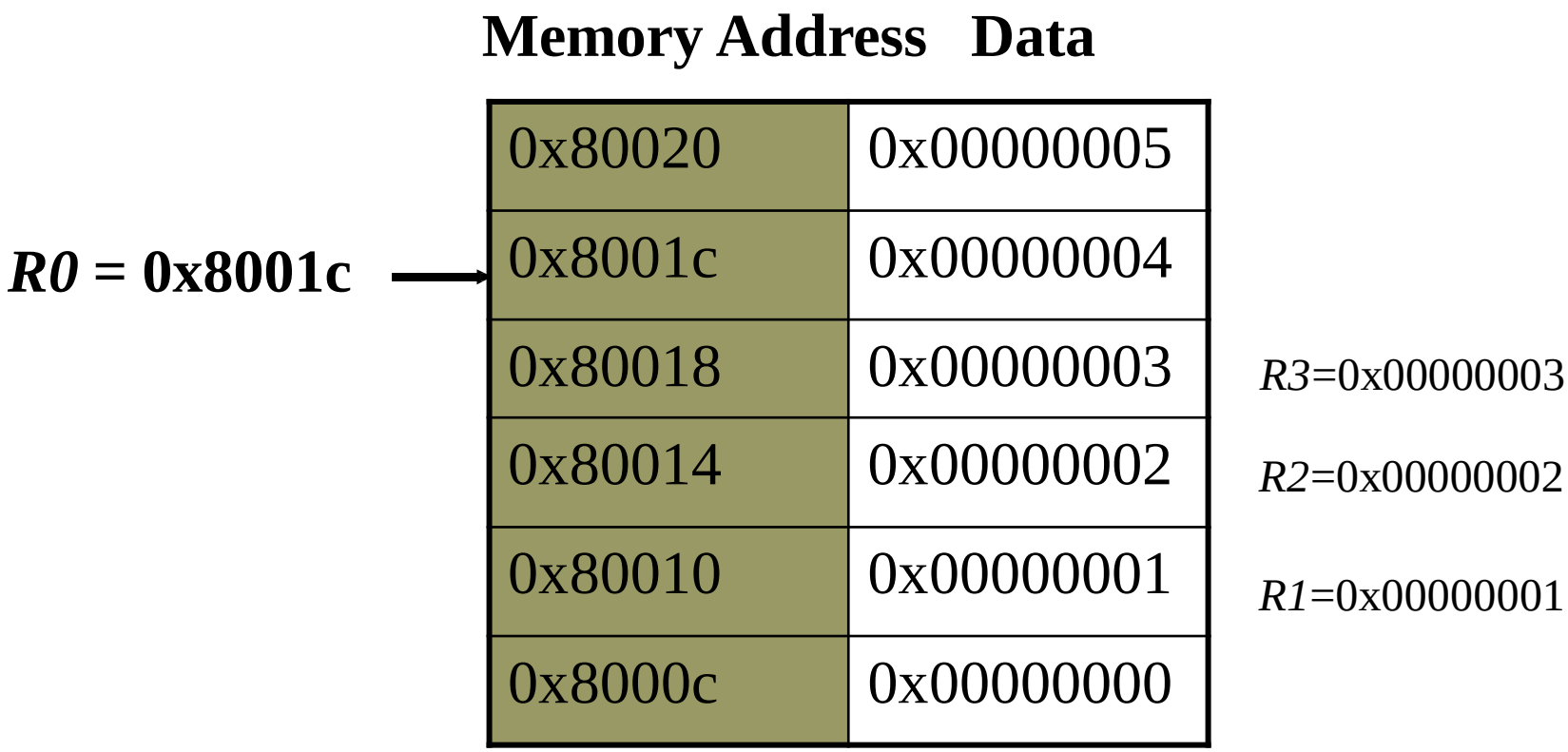


Figure 2

Multiple-Register Transfer (Cont.)

- ***Example 9 (Cont.)***
  - **POST:**

    r0 = 0x0008001c,

    r1 = 0x00000001,

    r2 = 0x00000002,

    r3 = 0x00000003

Multiple-Register Transfer (Cont.)

- ***Example 10***
  - PRE: as shown in Fig. 1
  - LDMIB r0!, {r1-r3}
  - **POST:**

    r0 = 0x0008001c

    r1 = 0x00000004

    r2 = 0x00000003

    r3 = 0x00000002

Post-Condition for LDMIB Instruction

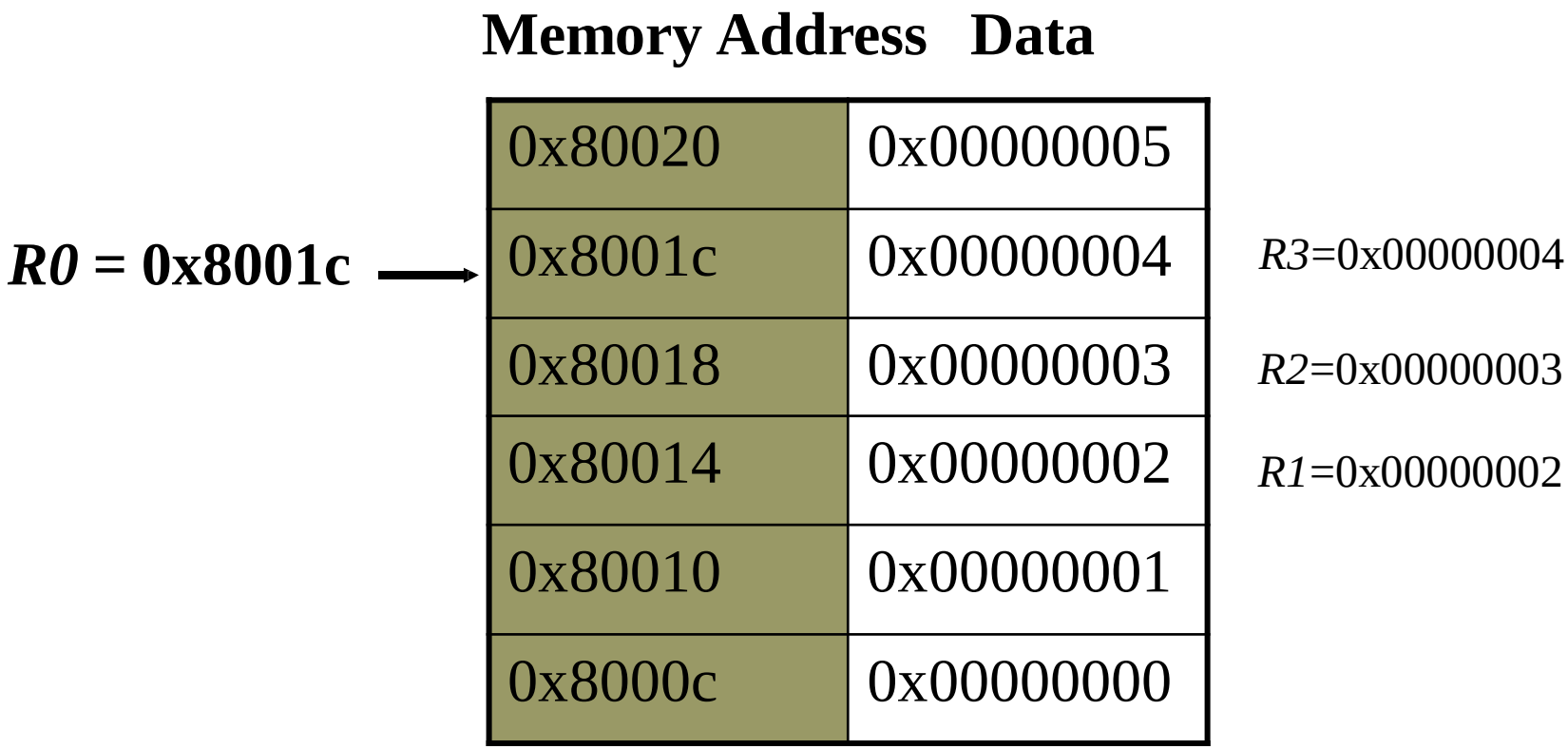


Figure 3

Multiple-Register Transfer (Cont.)

- Load-store multiple pairs when base update used (!)
  - Useful for saving a group of registers and store them later

Store multiple	**Load multiple**
STMIA	LDMDB
STMIB	LDMDA
STMDA	LDMIB
STMDB	LDMIA

Multiple-Register Transfer (Cont.)

- ***Example 11***
  - **PRE:**

    r0 = 0x00009000

    r1 = 0x00000009,

    r2 = 0x00000008

    r3 = 0x00000007
  - STMIB r0!, {r1-r3} MOV r1, #1 MOV r2, #2, MOV r3, #3

Multiple-Register Transfer (Cont.)

- ***Example 11 (Cont.)***
  - **PRE (2):**

    r0 = **0x0000900c**
    r1 = 0x00000001,
    r2 = 0x00000002
    r3 = 0x00000003
  - LDMDA r0!, {r1-r3}
  - **POST:**

    r0 = 0x00009000
    r1 = 0x00000009,
    r2 = 0x00000008
    r3 = 0x00000007

Multiple-Register Transfer (Cont.)

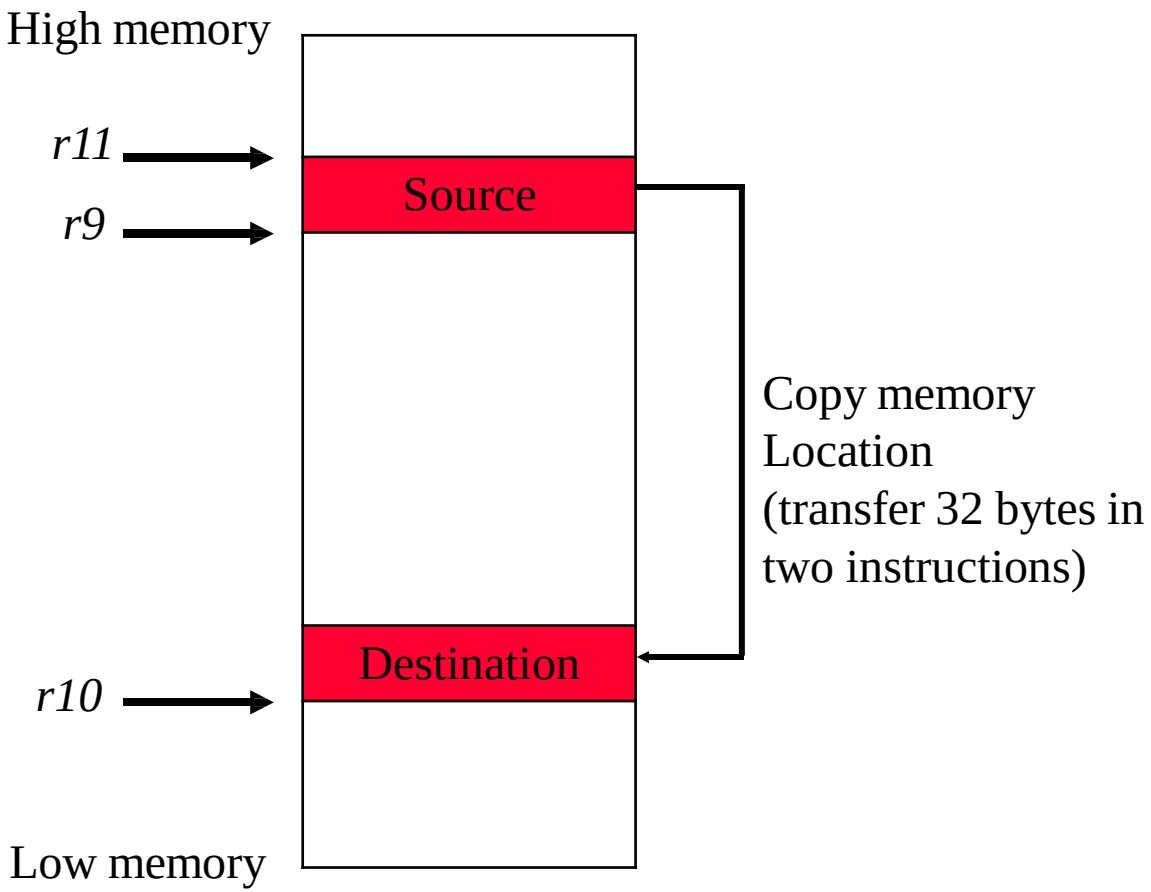
- ***Example 11 (Cont.)***
  - The STMIB stores the values 7, 8, 9 to memory
  - Then corrupt register *r1* to *r3* by MOV instruction
  - Finally, the LDMDA
    - Reloads the original values, and
    - Restore the base pointer *r0*

Multiple-Register Transfer (Cont.)

- ***Example 12***: the use of the load-store multiple instructions with *a block memory copy*

```
;r9 points to start of source data
;r10 points to start of destination data
;r11 points to end of the source
loop
    LDMIA   r9!, {r0-r7}    ;load 32 bytes from source and update r9
    STMIA   r10!, {r0-r7}   ;store 32 bytes to desti. and update r10
    CMP     r9, r11         ;have we reached the end
    BNE     loop
```

Multiple-Register Transfer (Cont.)



6.3.4 Stack Operations

- ARM architecture uses the *load-store multiple instruction* to carry out *stack operations*
  - ***PUSH***: use a *store multiple* instruction
  - ***POP***: use a *load multiple* instruction
- Stack
  - *Ascending* (A): stack grows towards higher memory addresses
  - *Descending* (D): stack grows towards lower memory addresses

6.3.4 Stack Operations (Cont.)

- Stack
  - *Full stack* (F): stack pointer *sp* points to the last valid item pushed onto the stack
  - *Empty stack* (E): *sp* points *after* the last item on the stack
    - The free slot where the next data item will be placed
- There are a number of aliases available to support stack operations
  - See next page

6.3.4 Stack Operations (Cont.)

- ARM support all four forms of stacks
  - ***Full ascending (FA)*:** grows up; base register points to the highest address containing a valid item
  - ***Empty ascending (EA)***: grows up; base register points to the first empty location
  - ***Full descending (FD)***: grows down; base register points to the lowest address containing a valid data
  - ***Empty descending (ED)***: grows down; base register points to the first empty location below the stack

Addressing Methods for Stack Operations

Addressing mode	**Description**	**Pop**	**=LDM**	**Push**	**=STM**
FA	Full ascending	LDMFA	LDMDA	STMFA	STMIB
FD	Full descending	LDMFD	LDMIA	STMFD	STMDB
EA	Empty ascending	LDMEA	LDMDB	STMEA	STMIA
ED	Empty descending	LDMED	LDMIB	STMED	STMDA

6.3.4 Stack Operations (Cont.)

- ***Example 13***
  - **PRE:**
    - r1 = 0x00000002
    - r4 = 0x00000003
    - sp = 0x00080014
  - STMFD sp!, {r1, r4}
  - **POST:**
    - r1 = 0x00000002
    - r4 = 0x00000003
    - sp = 0x0008000c

6.3.4 Stack Operations (Cont.)

- ***Example 13 (Cont.)***
  - STMFD – full stack push operation

PRE

Address	Data
0x80018	0x00000001
0x80014	0x00000002
0x80010	Empty
0x8000c	Empty

sp → 0x80014

POST

Address	Data
0x80018	0x00000001
0x80014	0x00000002
0x80010	0x00000003
0x8000c	0x00000002

sp → 0x8000c

6.3.4 Stack Operations (Cont.)

- **Example 14**
  - **PRE:**
    - r1 = 0x00000002
    - r4 = 0x00000003
    - sp = 0x00080010
  - STMED sp!, {r1, r4}
  - **POST:**
    - r1 = 0x00000002
    - r4 = 0x00000003
    - sp = 0x00080008

6.3.4 Stack Operations (Cont.)

- ***Example 14 (Cont.)***
  - STMED – empty stack push operation

PRE

Address	Data
0x80018	0x00000001
0x80014	0x00000002
0x80010	Empty
0x8000c	Empty
0x80008	Empty

sp →

POST

Address	Data
0x80018	0x00000001
0x80014	0x00000002
0x80010	0x00000003
0x8000c	0x00000002
0x80008	Empty

sp →

6.3.3 SWAP Instruction

- A special case of a load-store instruction
  - Swap the contents of memory with the contents of a register
  - An *atomic operation*
    - Cannot not be interrupted by any other instruction or any other buy access
    - The system “holds the bus” until the transaction is complete
    - Useful when implementing *semaphores* and *mutual exclusion* in an operating system

6.3.3 SWAP Instruction (Cont.)

- Syntax: SWP{B}{<cond>}Rd, Rm, [Rn]
  - *tmp = mem32[Rn]*
  - *Mem32[Rn] = Rm*
  - *Rd = tmp*
- **SWP:** swap a word between memory and a register
- **SWPB:** swap a byte between memory and a register

6.3.3 SWAP Instruction (Cont.)

- ***Example 15***
  - **PRE:**
    - Mem32[0x9000] = 0x12345678
    - r0 = 0x00000000
    - r1 = 0x11112222
    - r2 = 0x00009000
  - SWP r0, r1, [r2]
  - **POST:**
    - mem32[0x9000] = 0x11112222
    - r0 = 0x12345678
    - r1 = 0x11112222
    - r2 = 0x00009000

6.3.3 SWAP Instruction (Cont.)

- ***Example 15 (Cont.)***

```
SPIN
    MOV   r1, =semaphore
    MOV   r2, #1
    SWP   r3, r2, [r1]   ;hold the bus until complete
    CMP   r3, #1
    BEQ   spin
```

- The address pointed by the semaphore either contains the value of 1 or 0
- When semaphore value == 1 , loop until semaphore becomes 0 (updated by the holding process)

6.4 Software Interrupt Instruction

- SWI: software interrupt instruction
  - Cause a software interrupt exception
  - Provide a mechanism for applications to call operating system routines
  - Each SWI instruction has an associated SWI number
    - Used to represent a particular function call or routines

6.4 Software Interrupt Instruction (Cont.)

- Syntax: SWI{<cond>}SWI_number
  - *lr_svc = address of instruction following the SWI*
  - *spsr_svc = cpsr*
  - *pc = vector table + 0x8 ; jump to the swi handling*
  - *cpsr mode = SVC*
  - *cpsr I = 1 (mask IRQ interrupt)*

6.4 Software Interrupt Instruction (Cont.)

- ***Example 16***
  - **PRE:**
    - cpsr = nzcVqift_USER
    - pc = 0x00008000
    - lr = r14 = 0x003fffff
  - 0x00008000 SWI 0x123456
  - **POST:**
    - cpsr = nzcVqI**ft_SVC**
    - spsr = nzcVqift_USER
    - pc = 0x00000008
    - lr = 0x00008004

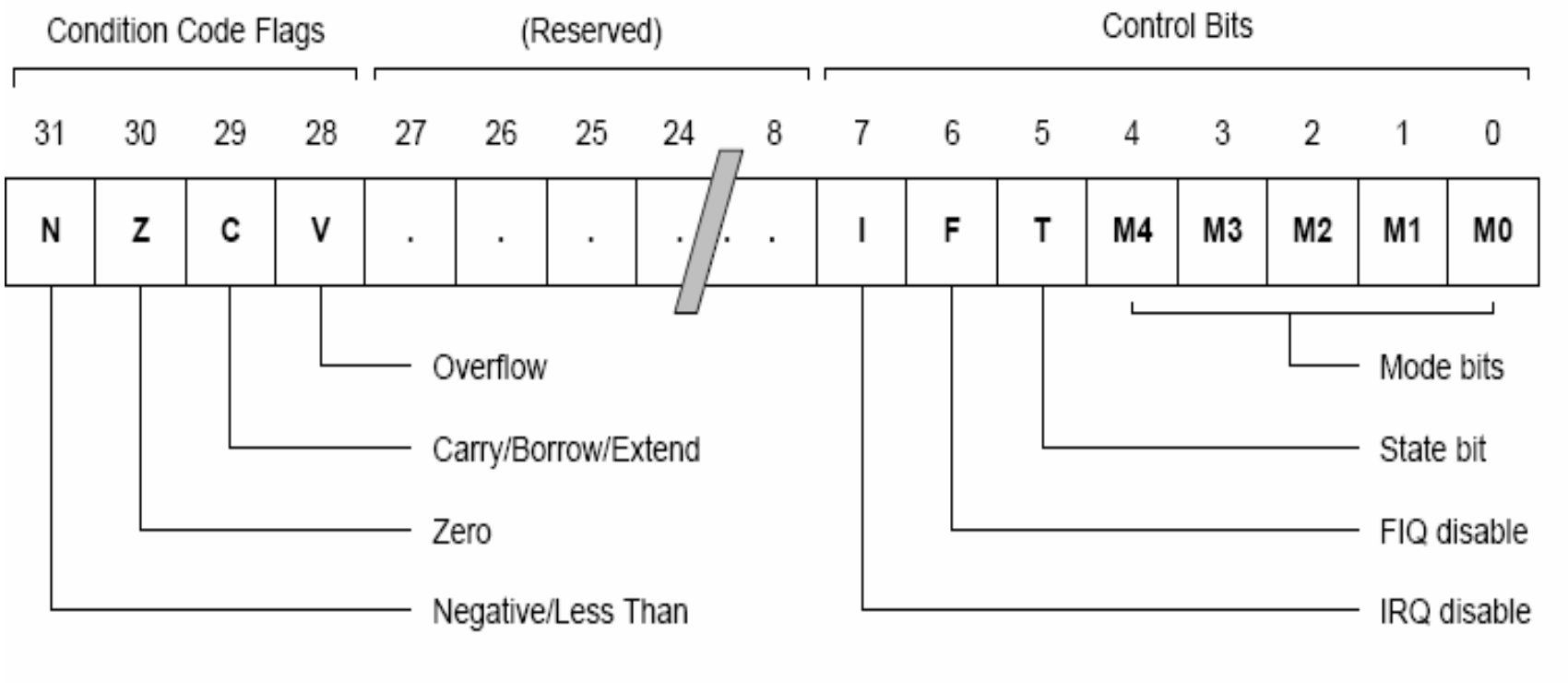
6.5 Program Status Register Instructions

- MRS
  - Transfer the contents of either the *cpsr* or *spsr* into a *register*
- MSR
  - Transter the contents of a *register* into the *cpsr* or *spsr*

6.5 Program Status Register Instructions (Cont.)

- Syntax
  - MRS{<cond>}Rd, <cpsr|spsr>
  - MSR{<cond>}<cpsr|spsr>_<fields>, Rm
  - MSR{<cond>}<cpsr|spsr>_<fields>, #immediate
- Field: any combination of
  - Flags: [24:31]
  - Status: [16:23]
  - eXtension[8:15]
  - Control[0:7]

PSR Registers



6.5 Program Status Register Instructions (Cont.)

- Note: You cannot access the SPSR in User or System Mode
  - Assembler cannot warn you because it does not know which mode will be executed in

6.5 Program Status Register Instructions (Cont.)

- ***Example 17***
  - **PRE:**
    - cpsr = nzcvqIFt_SVC
  - MRS r1, cpsr
  - BIC r1, r1, #0x80 ;0b10000000, clear bit 7
  - MSR cpsr_c, r1 ;enable IRQ interrupts
  - **POST:**
    - cpsr = nzcvq**i**Ft_**SVC**
  - Note that, this example must be in *SVC* mode
    - In user mode, you can only read all *cpsr* bits and can only update the *condition flag field f*, i.e., cpsr[24:31]

6.6 Conditional Execution

- Almost all ARM instruction can include an *optional condition code*
  - Instruction is only executed if the condition code *flags in the CPSR* meet the specified condition
  - The default is AL, or *always execute*
- Conditional executions depends on two components
  - The *condition field*: located in the instruction
  - The *condition flags*: located in the *cpsr*

Conditional Execution (Cont.)

- ***Example 18***

```
ADDEQ   r0, r1, r2
; r0 = r1 + r2 if zero flag is set
```

Condition Codes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS/HS	C set	Higher or same (unsigned >=)
CC/LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned <=)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V different	Signed <
GT	Z clear, and N and V the same	Signed >
LE	Z set, or N and V different	Signed <=
AL	Any	Always (usually omitted)

6.6 Conditional Execution (Cont.)

- Thus, before activate conditional execution
  - *There must be an instruction that updates the conditional code flag according the result*
  - If not specified, instructions will not update the flags
- To make an instruction update the flags
  - Include the *S* suffix
  - Example: `ADDS r0, r1,r2`

6.6 Conditional Execution (Cont.)

- However, some instructions always update the flags
  - Do not require the *S* suffix
  - CMP, CMN, TST, TEQ
- Flags are preserved until updated
- Thus, you can execute an instruction conditionally, based upon the flags set *in another instruction*, either:
  - Immediately after the instruction which updated the flags
  - After any number of intervening instructions that have not updated the flags.

6.6 Conditional Execution (Cont.)

- ***Example 18***
  - Transfer the following code into the assembly language
  - Assume *r1* = a, *r2* = b

```
while ( a!=b )
{
        if (a >b) a -=b; else b -=a;
 }
```

6.6 Conditional Execution (Cont.)

- ***Example 18: Solution 1***

```
gcd
        CMP     r1, r2
        BEQ     complete
        BLT     lessthan
        SUB     r1, r1, r2
        B       gcd
lessthan
        SU      r2, r2, r1
        B  B    gcd
complete
```

6.6 Conditional Execution (Cont.)

- ***Example 18: Solution 2***

```
gcd
        CMP     r1, r2
        SUBGT   r1,  r1,
        T       r2   r2,
        SUBLT   r2,   r1
        BNE     gcd
```

- Solution 2 dramatically reduces the number of instructions !!!