Amrita School Of Engineering, Bangalore Campus

INSTRUCTION SET LECTURE - 4

PREVIOUSLY

➤ Data Processing Instructions

TODAY…

➤ Data Processing Instructions – Contd…

CPSR FLAGS

- ➤ Carry/~Borrow

  - ➤ In unsigned arithmetic, watch the carry flag to detect errors.

  - ➤ Bit goes from MSB

- ➤ Overflow

  - ➤ In signed arithmetic, watch the overflow flag to detect errors.

  - ➤ Will be set if an overflow occurs into bit 31 of the result

- ➤ Negative

  - ➤ Set to the logical value of bit 31 of the result

- ➤ Zero

  - ➤ Sets only if result is zero

CONTD…

➤ Logical Instructions

  ➤ V Flag unaffected

  ➤ C flag will be set to the carry out from the barrel shifter

  ➤ Z flag will be set if and only if the result is all zeros

  ➤ N flag will be set to the logical value of bit 31 of the result.

CONTD…

➤ Arithmetic Instructions

   ➤ V/C flag based on Signed/Unsigned

   ➤ Z flag will be set if and only if the result was zero

   ➤ N flag will be set to the value of bit 31 of the result

FLAGS FOR MUL INSTRUCTIONS

➤ The N (Negative) and Z (Zero) flags are set correctly on the result

  ➤ N is made equal to bit 31 of the result

  ➤ Z is set if and only if the result is zero.

➤ The C (Carry) flag is set to a meaningless value

➤ V (oVerflow) flag is unaffected.

INSTRUCTION FORMAT

➤ Ex: ADD r5,r1,r2;

14	0	0	4	0	1	5	2
4-bits Cond	2 bits F	1 bit I	4 bits Opcode	1 bit S	4 bits Rn	4 bits Rd	12 bits Operand2

➤

➤

➤ Operand2 second source operand

➤ I If 0, second source is a register else second source is 12-bit imm.

➤ S set cond code

➤ F Instrn format, 0 = data processing instrn format

CONTD…

➤ What ARM instruction does this represent?

14	0	0	4	0	0	1	2
4-bits Cond	2 bits F	1 bit I	4 bits Opcode	1 bit S	4 bits Rn	4 bits Rd	12 bits Operand2

ADD R1, R0, R2

LOGICAL OPERATIONS

- ➤ AND ; &

- ➤ ORR ; |

- ➤ MVN ; ~ MOV

- ➤ LSL ; <<

- ➤ LSR ; >>

CONTD…

➤ Shift the second operand:

➤ ADD r5, r1, r2, LSL #2;

➤ r5 = r1 + (r2<<2)

CONTD…

➤ Shift right r5 by 4-bits and place the result in r6?

MOV r6,r5, LSR #4;

 MOV r6,r5, LSR r3;

   ➤ r6 = r5 >> r3

CONTD…

➤ Instruction format??

31-28	27-26	25	24-21	20	19-16	15-12	11-0
4-bits Cond	2 bits F	1 bit I	4 bits Opcode	1 bit S	4 bits Rn	4 bits Rd	12 bits Operand2

➤ 12-bits operand2 field is interpreted as:

11	10	9	8	7	6	5	4	3	2	1	0
Shift_imm					Shift		0	Rm			
Rs				0	Shift		1	Rm			

Shift = Type of shift

4ᵗʰ bit = 0 Shift_amt = Imme

4ᵗʰ bit = 1 Shift_amt = Rs

SUMMARY

| Opcode (124:21)	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd:=RnANDOp2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn 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	ScconRnANDOp2
1001	TEQ	Test equivalence	Sec on Rn EOR Op2
1010	CMP	Compare	Sec on Rn - Op2
1011	CMN	Compare negated	Sec on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd:=RnANDNOTOp2
1111	MVN	Move negated	Rd:=NOTOp2

LOAD / STORE INSTRUCTIONS

➤ **The ARM is a Load / Store Architecture:**

  ➤ Does not support memory to memory data processing operations.

  ➤ Must move data values into registers before using them.

➤ **This might sound inefficient, but in practice isn't:**

  ➤ Load data values from memory into registers.

  ➤ Process data in registers using a number of data processing instructions which are not slowed down by memory access.

  ➤ Store results from registers out to memory.

➤ **The ARM has three sets of instructions which interact with main memory. These are:**

  ➤ Single register data transfer (LDR / STR).

  ➤ Block data transfer (LDM/STM).

  ➤ Single Data Swap (SWP).

SINGLE REGISTER DATA TRANSFER

➤ The basic load and store instructions are:

    ➤ Load and Store Word or Byte

        ➤ LDR / STR / LDRB / STRB

➤ ARM Architecture Version 4 also adds support for halfwords and signed data.

    ➤ Load and Store Halfword

        ➤ LDRH / STRH

    ➤ Load Signed Byte or Halfword - load value and sign extend it to 32 bits.

        ➤ LDRSB / LDRSH

    ➤ STRSB/STRSH is not available.

➤ All of these instructions can be conditionally executed by inserting the appropriate condition code after STR / LDR.

    ➤ e.g. LDREQB

➤ Syntax:

    ➤ <LDR|STR>{<cond>}{<size>} Rd, <address>

➤ The memory location to be accessed is held in a base register

  ➤ STR r0, [r1] ; Store contents of r0 to location pointed to by contents of r1.

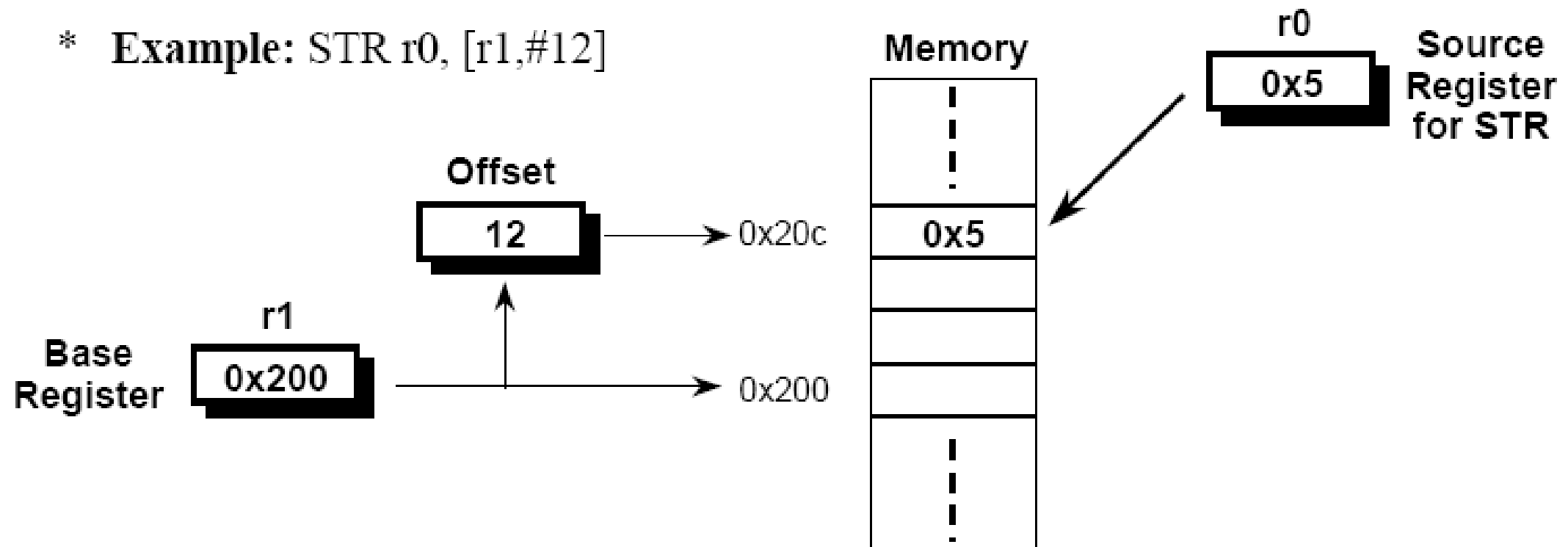 ➤ LDR r2, [r1] ; Load r2 with contents of memory location pointed to by contents of r1.
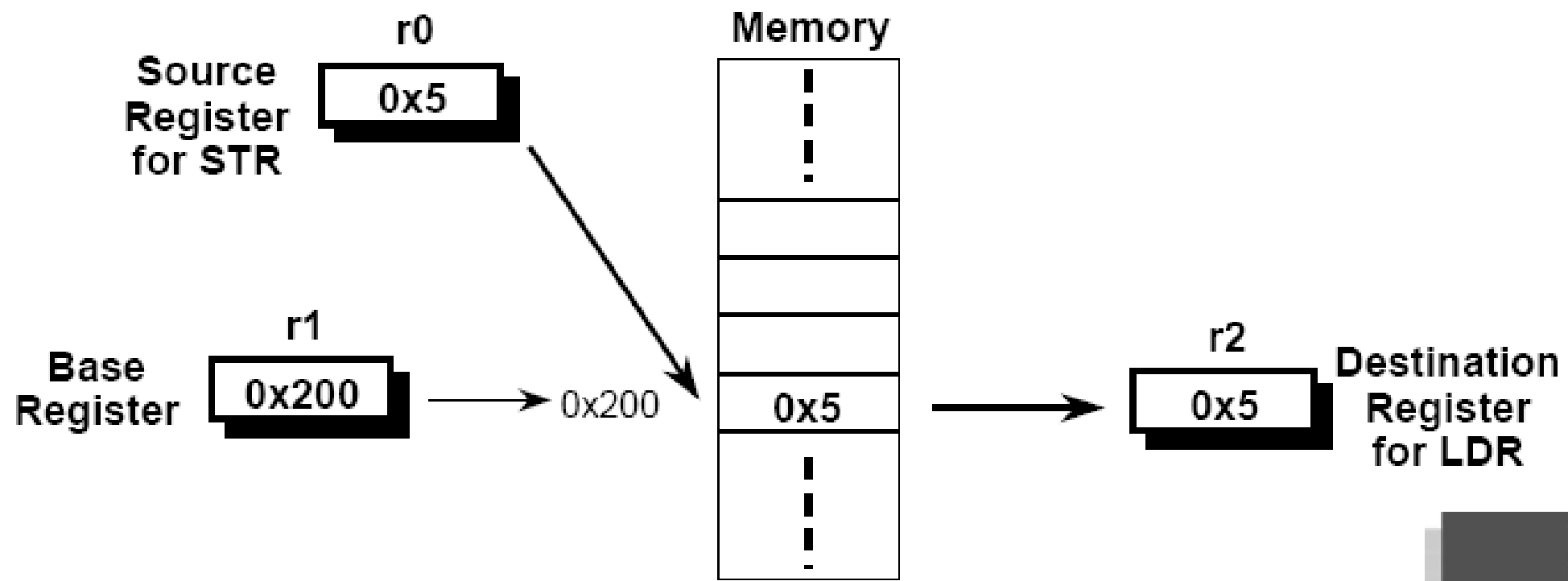
➤ As well as accessing the actual location contained in the base register, these instructions can access a location offset from the base register pointer.

➤ This offset can be

  ➤ An unsigned 12bit immediate value (i.e. 0 - 4095 bytes).

  ➤ A register, optionally shifted by an immediate value

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

  ➤ Prefix the offset value or register with '+' (default) or '-'.

➤ This offset can be applied:

  ➤ before the transfer is made: *Pre-indexed addressing*

    ➤ optionally *auto-incrementing the base register, by postfixing the* instruction with an '!'.

  ➤ after the transfer is made: *Post-indexed addressing*

    ➤ causing the base register to be *auto-incremented.*
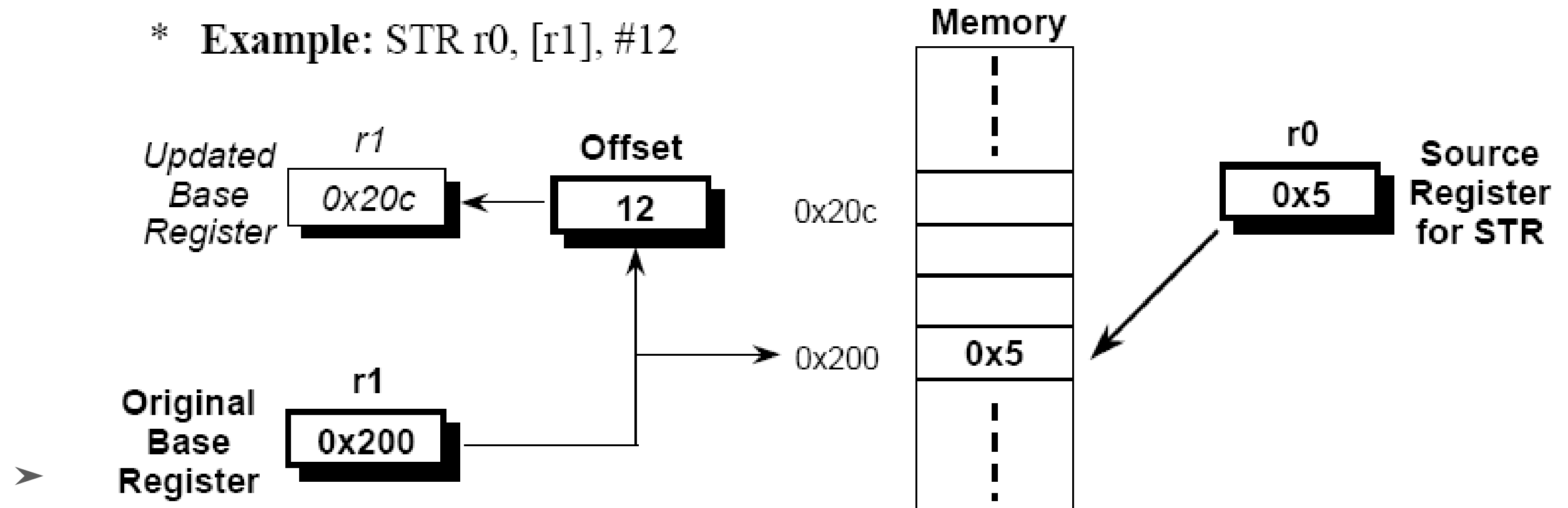
LOAD AND STORE WORD OR BYTE:PRE-INDEXED ADDRESSING

* **Example:** STR r0, [r1,#12]

Memory

r0 → Source Register for STR
0x5

Offset
12 → 0x20c
0x5

r1
Base Register
0x200 → 0x200

- ➤ To store to location 0x1f4 instead use: STR r0, [r1,#-12]

- ➤ To auto-increment base pointer to 0x20c use: STR r0, [r1, #12]!

- ➤ If r2 contains 3, access 0x20c by multiplying this by 4:

  - ➤ STR r0, [r1, r2, LSL #2]

* **Example:** STR r0, [r1], #12

➤ STR r0, [r1], #-12

➤ If r2 contains 3, auto-increment base register to 0x20c by multiplying this by 4:

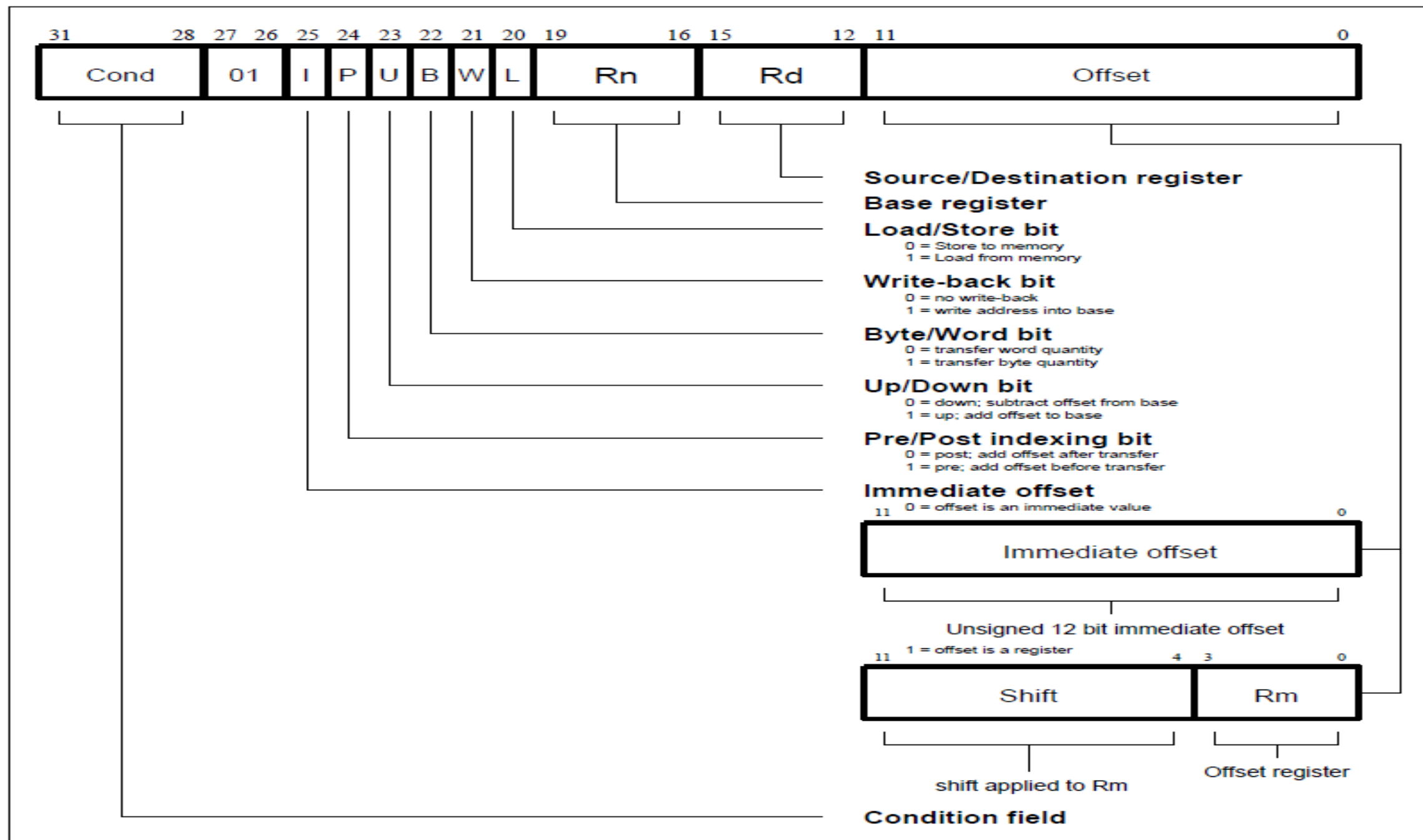 ➤ STR r0, [r1], r2, LSL #2

INCSTRUCTION ENCODING

Figure 4-14: Single data transfer instructions

USE OF PC

➤ Write-back must not be specified if R15 is specified as the base register (Rn).

➤ R15 must not be specified as the register offset (Rm).

EXAMPLE USAGE OF ADDRESSING MODES

➤ Imagine an array, the first element of which is pointed to by the contents of r0.

➤ If we want to access a particular element, then we can use pre-indexed addressing:

  ➤ r1 is element we want.

  ➤ LDR r2, [r0, r1, LSL #2]

➤ If we want to step through every element of the array, for instance to produce sum of elements in the array, then we can use post-indexed addressing within a loop:

  ➤ r1 is address of current element (initially equal to r0).

  ➤ LDR r2, [r1], #4

➤ Use a further register to store the address of final element, so that the loop can be correctly terminated.

OFFSETS FOR HALFWORD AND SIGNED HALFWORD / BYTE ACCESS

➤ The Load and Store Halfword and Load Signed Byte or Halfword instructions can make use of pre- and post-indexed addressing in much the same way as the basic load and store instructions.

➤ However the actual offset formats are more constrained:

  ➤ The immediate value is limited to 8 bits (rather than 12 bits) giving an offset of 0-255 bytes.

  ➤ The register form cannot have a shift applied to it.

INSTRUCTION ENCODING – WITH REGISTER OFFSET
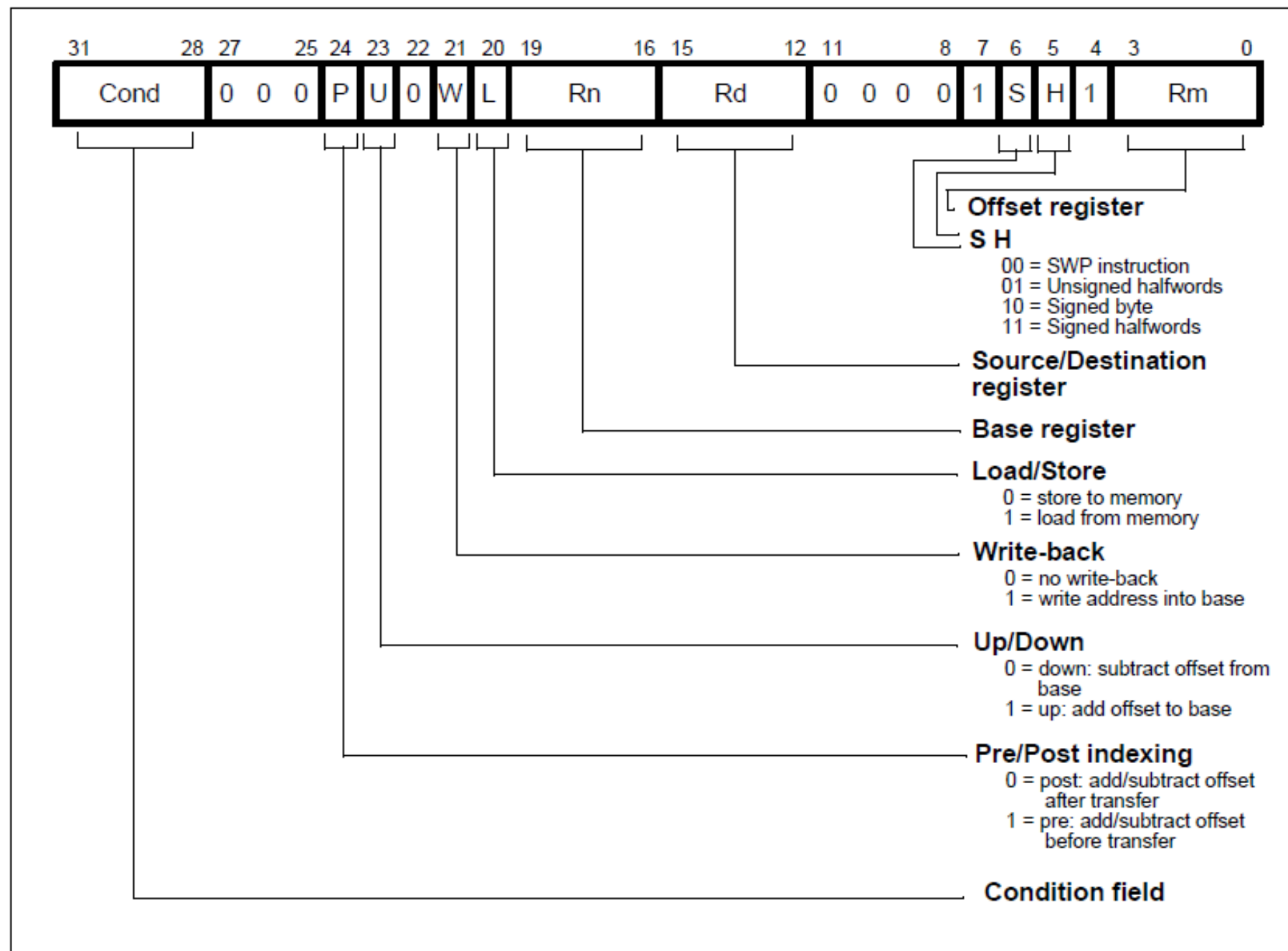
# INSTRUCTION ENCODING – WITH IMMEDIATE OFFSET
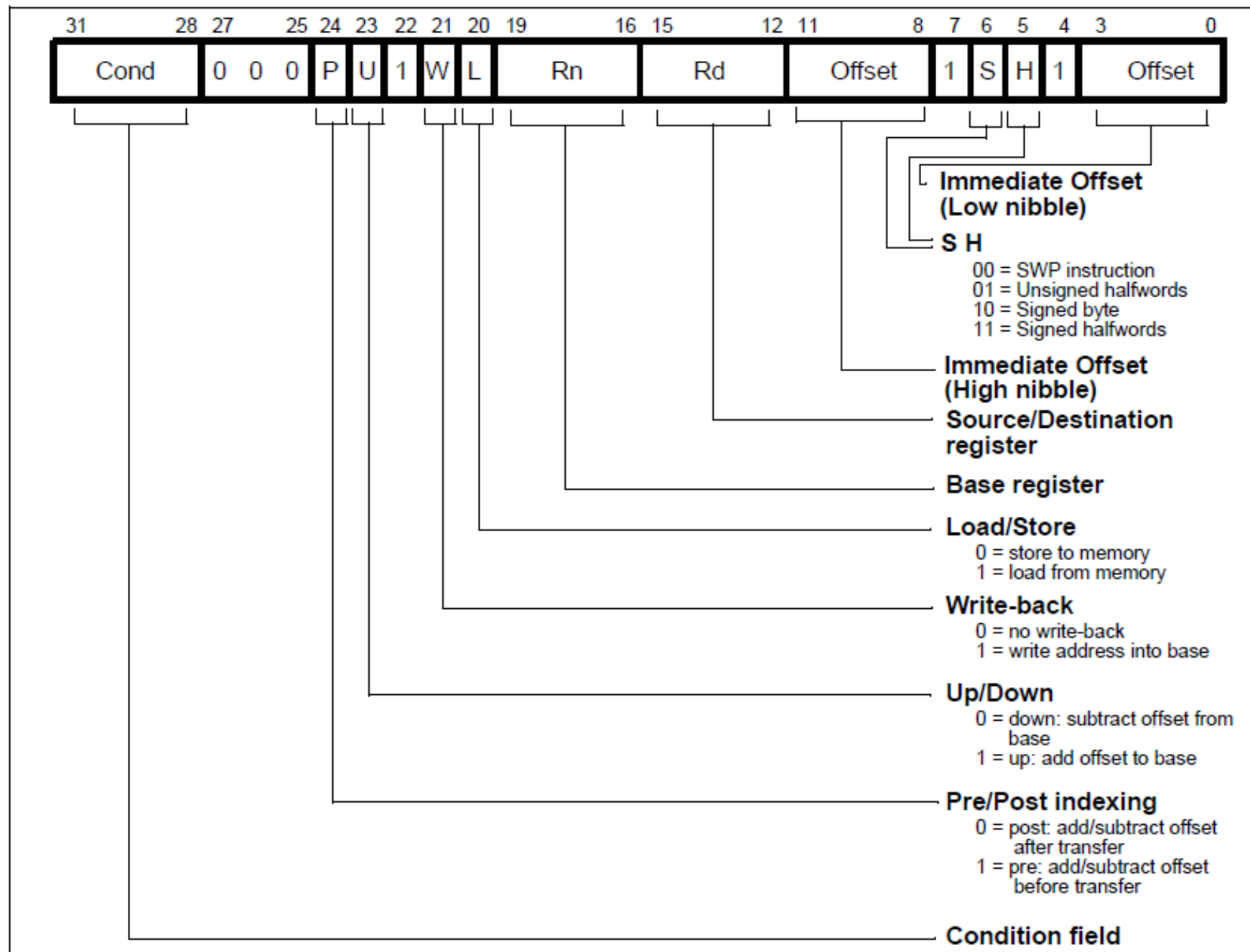
BLOCK DATA TRANSFER

➢ The Load and Store Multiple instructions (LDM / STM) allow between 1 and 16 registers to be transferred to or from memory.

➢ The transferred registers can be either:

   ➢ Any subset of the current bank of registers (default).

   ➢ Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a '^').

CONTD…

➤ 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
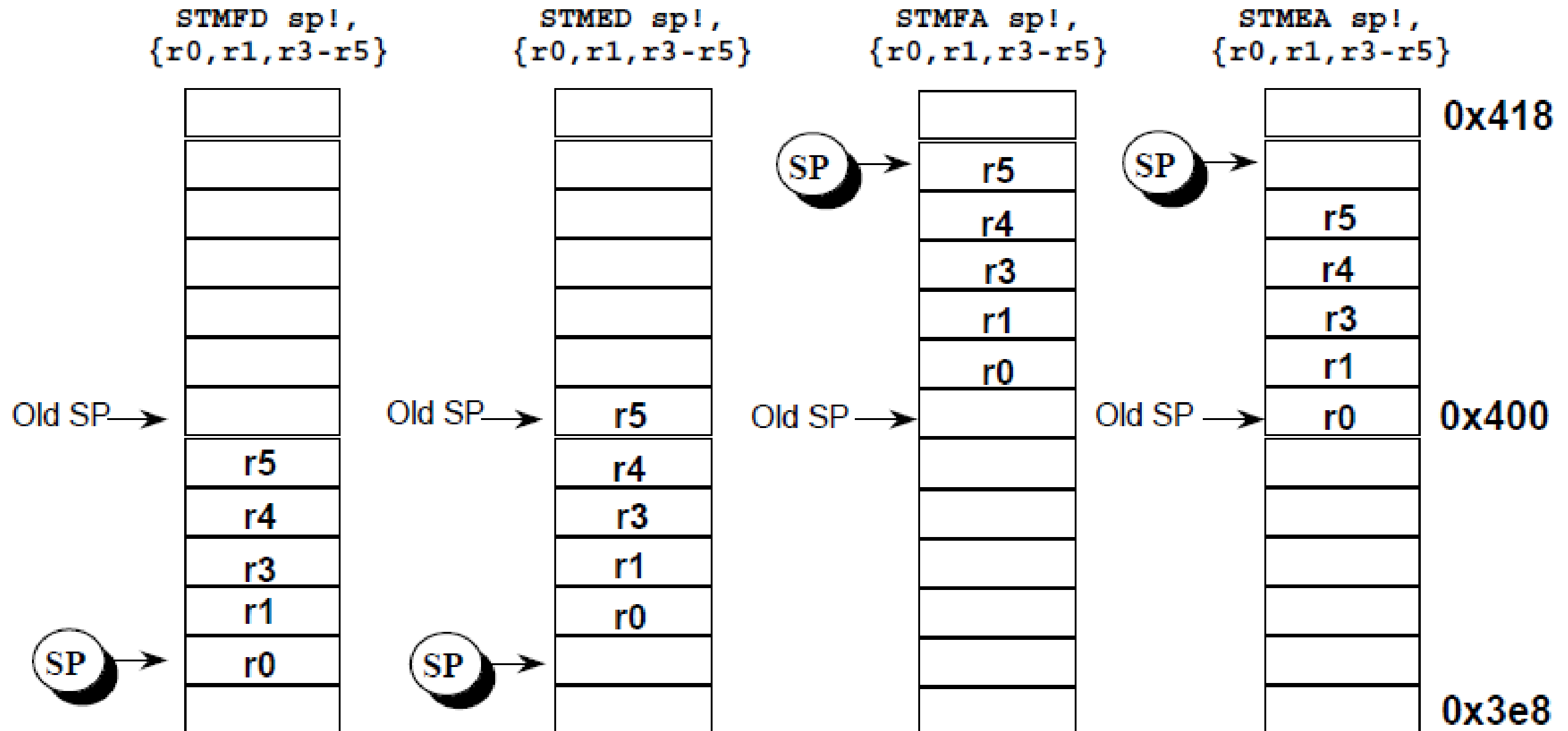
➤ When LDM / STM are not being used to implement stacks, it is clearer to specify exactly what functionality of the instruction is:

  ➤ i.e. specify whether to increment / decrement the base pointer, before or after the memory access.

➤ In order to do this, LDM / STM support a further syntax in addition to the stack one:

  ➤ STMIA / LDMIA : Increment After

  ➤ STMIB / LDMIB : Increment Before

  ➤ STMDA / LDMDA : Decrement After

  ➤ STMDB / LDMDB : Decrement Before

- Multiple register load and store instructions
  - enable transfer of large quantities of data
  - used for procedure entry and exit, to save/restore workspace registers, to copy blocks of data around memory

Multiple register data transfers

LDMIA r1, {r0, r2, r5}	$r0 := mem_{32}[r1]$ $r2 := mem_{32}[r1 + 4]$ $r5 := mem_{32}[r1 + 8]$

Note: any subset (or all) of the registers may be transferred with a single instruction

Note: the order of registers within the list is insignificant

Note: including r15 in the list will cause a change in the control flow

CONTD…

➤ 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.

→ r12 points to the start of the source data

→r14 points to the end of the source data

→r13 points to the start of the destination data

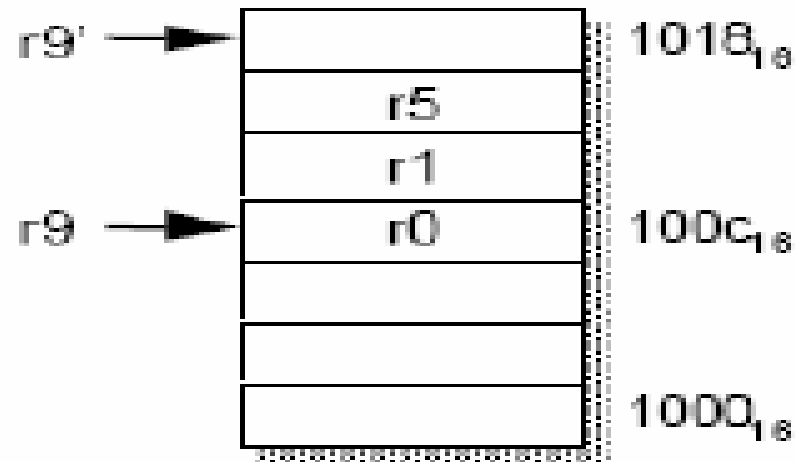loop   LDMIA r12!, {r0-r11} ; load 48 bytes

STMIA r13!, {r0-r11} ; and store them

CMP r12, r14 ; check for the end

BNE loop ; and loop until done
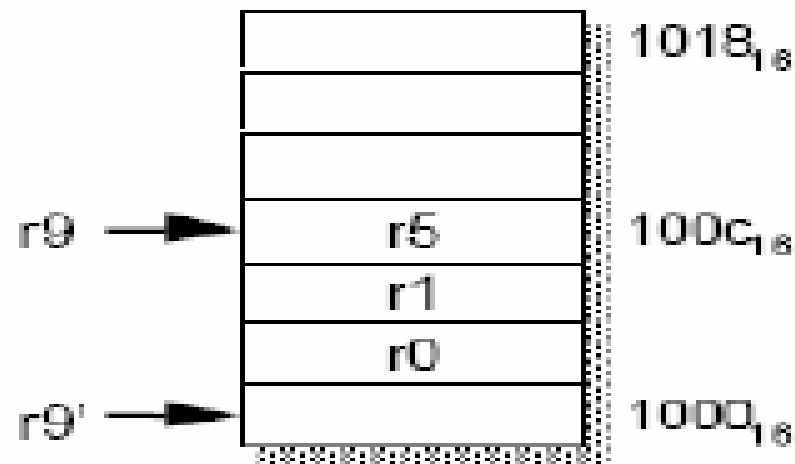
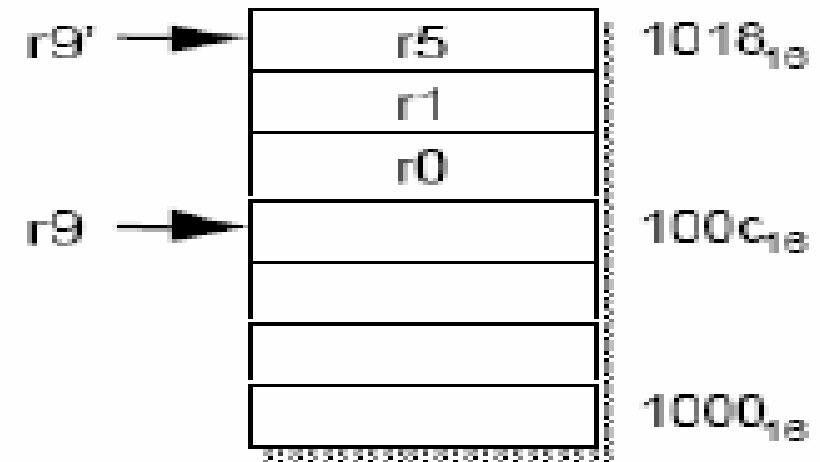➤ This loop transfers 48 bytes in 31 cycles

➤ Over 50 Mbytes/sec at 33 MHz

STMIA r9!, {r0,r1,r5}
Increment after

STMIB r9!, {r0,r1,r5}
Increment before

STMDA r9!, {r0,r1,r5}
Decrement after
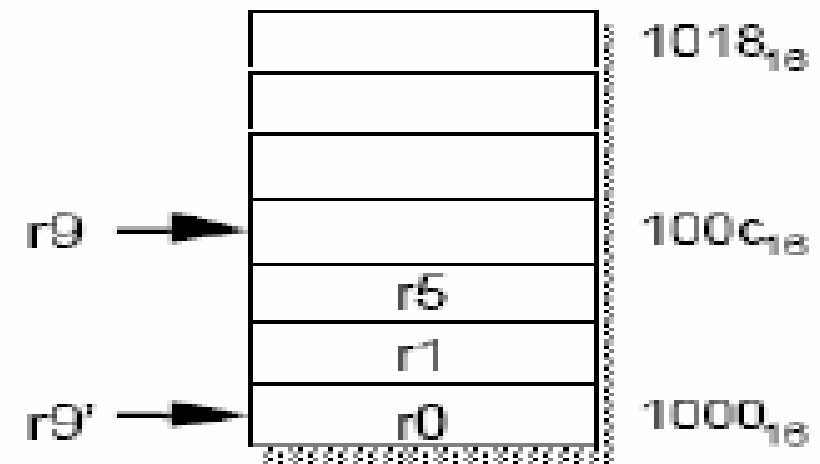
STMDB r9!, {r0,r1,r5}
Decrement before

USE OF THE S BIT

➢ When the S bit is set in a LDM/STM instruction its meaning depends on whether or not R15 is in the transfer list and on the type of instruction.

➢ The S bit should only be set if the instruction is to execute in a privileged mode.

➢ **LDM with R15 in transfer list and S bit set (Mode changes)**

  ➢ If the instruction is a LDM then SPSR_<mode> is transferred to CPSR at the same time as R15 is loaded.

- **STM with R15 in transfer list and S bit set (User bank transfer)**

  - The registers transferred are taken from the User bank rather than the bank corresponding to the current mode.

  - This is useful for saving the user state on process switches.

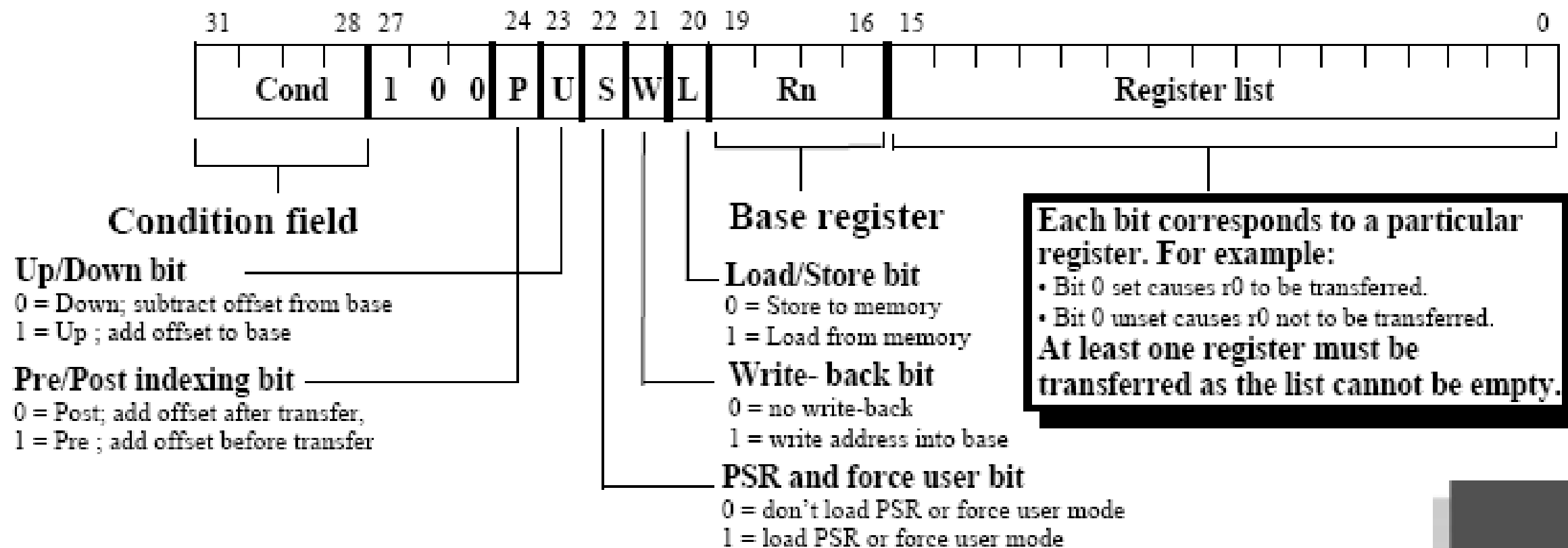 - Base write-back should not be used when this mechanism is employed.

CONTD…

➤ **R15 not in list and S bit set (User bank transfer)**

➤ For both LDM and STM instructions, the User bank registers are transferred rather than the register bank corresponding to the current mode.

➤ This is useful for saving the user state on process switches.

➤ Base write-back should not be used when this mechanism is employed.

CONTD…

- ➤ 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.

- ➤ These instructions are very efficient for

  - ➤ Saving and restoring context

    - ➤ For this useful to view memory as a stack.

  - ➤ Moving large blocks of data around memory

    - ➤ For this useful to directly represent functionality of the instructions.

INSTRUCTION FORMAT

Condition field

Up/Down bit
0 = Down; subtract offset from base
1 = Up ; add offset to base

Pre/Post indexing bit
0 = Post; add offset after transfer,
1 = Pre ; add offset before transfer

PSR and force user bit
0 = don't load PSR or force user mode
1 = load PSR or force user mode

Base register

Load/Store bit
0 = Store to memory
1 = Load from memory

Write-back bit
0 = no write-back
1 = write address into base

Each bit corresponds to a particular register. For example:
• Bit 0 set causes r0 to be transferred.
• Bit 0 unset causes r0 not to be transferred.
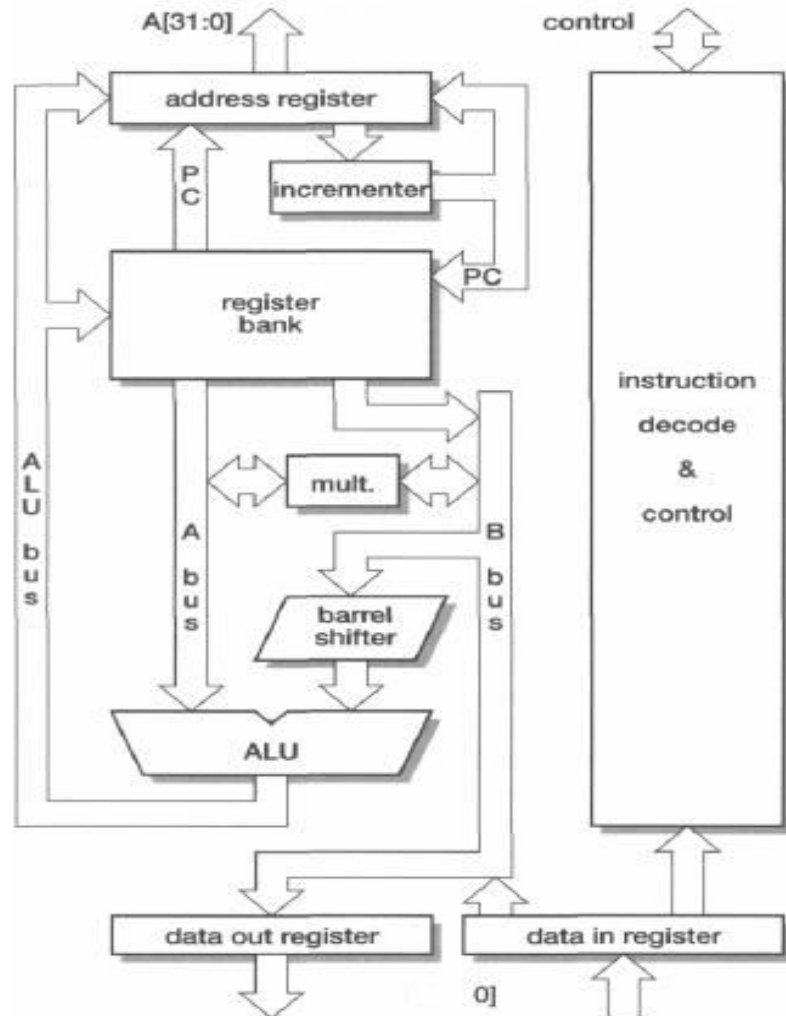At least one register must be transferred as the list cannot be empty.

THANK YOU

Today…

- 3 Stage Pipeline

- 5 Stage Pipeline

3 Stage Pipeline Organization

- Principle Components

  - Register Banks

    - Ports???

  - Barrel Shifters

  - ALU

  - Address Register and Incrementer

  - Data Register

  - Instruction Decoder and Control Logic

- Ins and data Memory???

Pipeline Stages

- **Fetch:**

  - The instruction is fetched from memory and placed in the instruction pipeline

- **Decode**

  - The instruction is decoded and the datapath control signals prepared for the next cycle.

  - In this stage the instruction 'owns' the decode logic but not the datapath.

- **Execute**
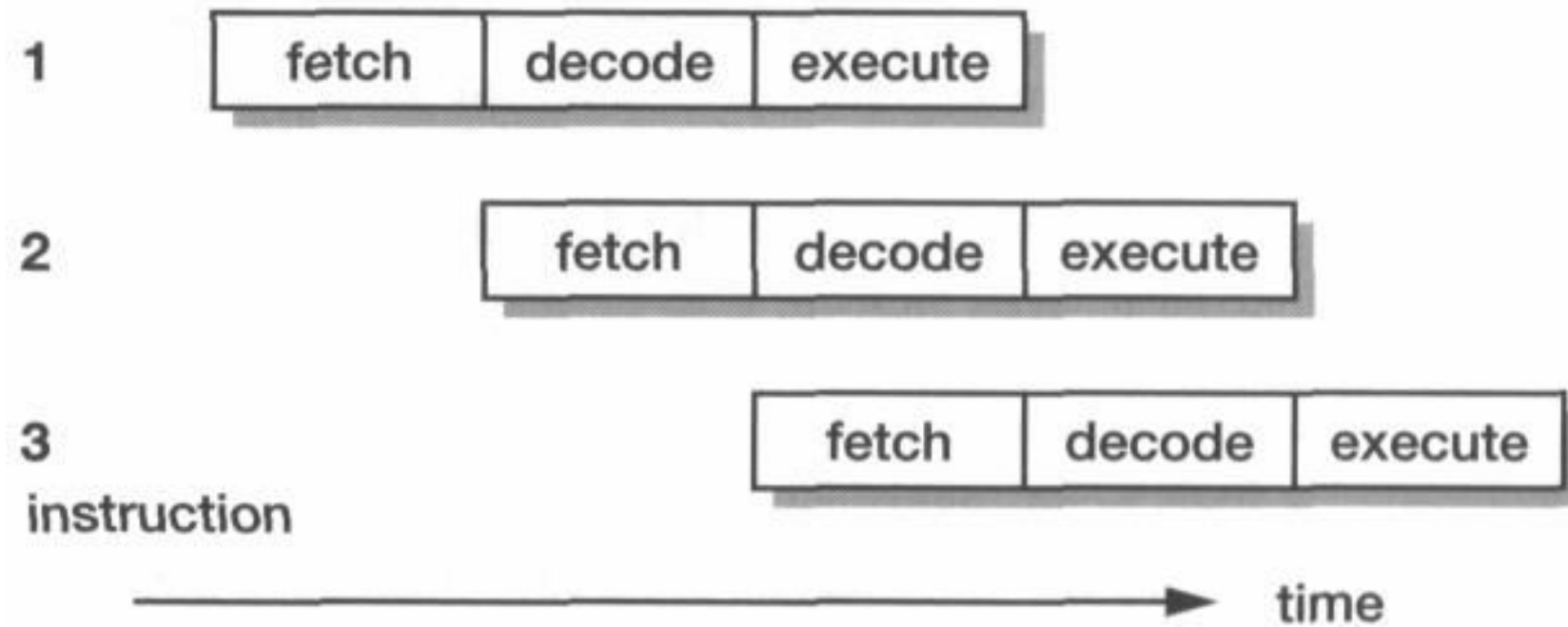
  - The instruction 'owns' the datapath;

  - The register bank is read, an operand shifted, the ALU result generated and written back into a destination register.

Pipeline Stages

- With simple single cycle data transfer instruction, each pipeline stage enables one instruction to be completed every clock cycle.

- So Latency is 3 clock cycles

- Throughput is 1 instruction per cycle.

Single Cycle Instruction

ARM single-cycle instruction 3-stage pipeline operation.

Multi-Cycle Instruction

1 | fetch ADD | decode | execute

2 | fetch STR | decode | calc. addr. | data xfer

3 | fetch ADD | decode | execute

4 | fetch ADD | decode | execute

5 | fetch ADD | decode | execute

instruction

time
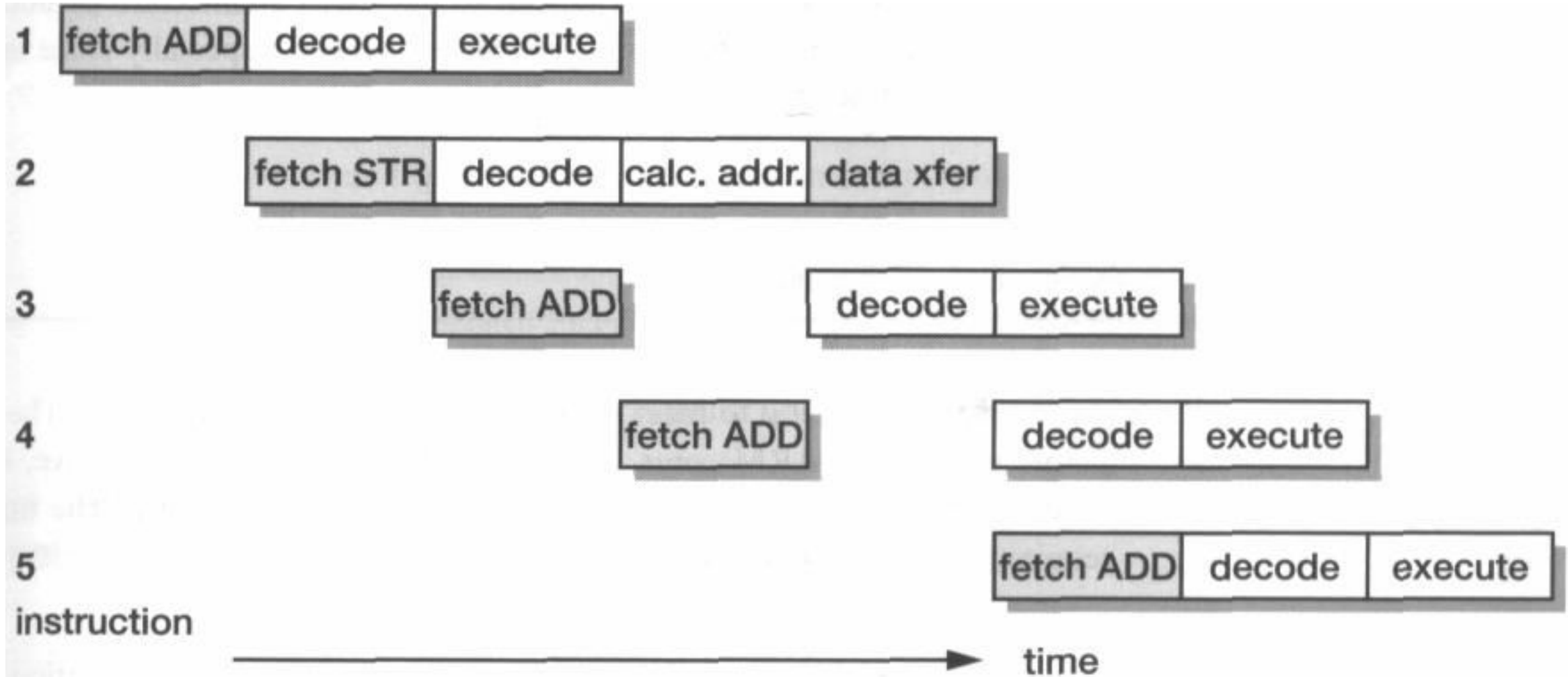
ARM multi-cycle instruction 3-stage pipeline operation.

Multi-Cycle Instruction

- All instructions occupy the datapath for one or more adjacent cycles.

- For each cycle that an instruction occupies the datapath, it occupies the decode logic in the immediately preceding cycle.

- During the first datapath cycle each instruction issues a fetch for the next instruction but one.

- Branch instructions flush and refill the instruction pipeline.

PC??

One consequence of the pipelined execution model used on the ARM is that the program counter, which is visible to the user as r!5, must run ahead of the current instruction. If, as noted above, instructions fetch the next instruction but one during their first cycle, this suggests that the PC must point eight bytes (two instructions) ahead of the current instruction.

This is, indeed, what happens, and the programmer who attempts to access the PC directly through r!5 must take account of the exposure of the pipeline here. However, for most normal purposes the assembler or compiler handles all the details.

Even more complex behaviour is exposed if r!5 is used later than the first cycle of an instruction, since the instruction will itself have incremented the PC during its first cycle. Such use of the PC is not often beneficial so the ARM architecture definition specifies the result as 'unpredictable' and it should be avoided, especially since later ARMs do not have the same behaviour in these cases.

3 Stage Pipeline

- The time, *T* , required to execute a given program is given by:

$$T_{prog} = \frac{N_{inst} \times CPI}{f_{clk}},$$

Equation 11

- How to decrease Tprog?

Fclk ↑ CPI ↓

3 Stage Pipeline

- Memory bottleneck

  - Von Neumann bottleneck

- A 3-stage ARM core accesses memory on (almost) every clock cycle either to fetch an instruction or to transfer data.

- Simply tightening up on the few cycles where the memory is not used will yield only a small performance gain.

- To get a significantly better CPI the memory system must deliver more than one value in each clock cycle either by delivering more than 32 bits per cycle from a single memory or by having separate memories for instruction and data accesses.

- What's the solution then?

  5 Stage Pipeline.

5 Stage Pipeline

- To meet the demand of high performance 3 stage pipeline is evolved into 5 stage in later version of ARM (after ARM7).

  - **Fetch**
  - **Decode**
  - **Execute**
  - **Buffer/Data Read**
  - **Write Back**

5 Stage Pipeline

- Fetch
  - The instruction is fetched from memory and placed in the instruction pipeline.

- Decode
  - The instruction is decoded and register operands read from the register file.
  - There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

- Execute
  - An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

5 Stage Pipeline

- Buffer/Data (read)
  - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.

- Write Back
  - The results generated by the instruction are written back to the register file, including any data loaded from memory.

5 Stage Pipeline

- This 5-stage pipeline has been used for many RISC processors and is considered to be the 'classic' way to design such a processor.

- Although the ARM instruction set was not designed with such a pipeline in mind, it maps onto it relatively simply.

    - There are three source operand read ports and two write ports in the register file (where a 'classic' RISC has two read ports and one write port), and the inclusion of address incrementing hardware in the execute stage to support load and store multiple instructions.

    We will describe the ARM 5 stage Pipeline organization after discussing the instruction execution.(Simple ------→ complex)

5 Stage Pipeline

- Note:
- The coming slides discusses the data path activity for instruction execution
  - Data processing instructions
  - Data transfer instructions
  - Branch instruction

  - However we discuss them with respect to either single clock cycle (data processing instructions) or 3-stage pipelining.

  - We are postponing the 5-stage pipeline organization for now.

ARM instruction Execution

Data Processing Instructions

- A data processing instruction requires two operands,
  - one of which is always a register
  - and the other is either a second register or an immediate value.

- The second operand is passed through the barrel shifter where it is subject to a general shift operation, then it is combined with the first operand in the ALU using a general ALU operation.

- Finally, the result from the ALU is written back into the destination register (and the condition code register may be updated).

- All these operations take place in a single clock cycle.

ARM instruction Execution

Data Processing Instructions

- The PC value in the address register is incremented and copied back into both the address register and r15 in the register bank, and the next instruction but one is loaded into the bottom of the instruction pipeline *(i. pipe).*

- The immediate value, when required, is extracted from the current instruction at the top of the instruction pipeline.

- For data processing instructions only the bottom eight bits (bits [7:0]) of the instruction are used in the immediate value.

ARM instruction Execution

Data Processing Instruction datapath activity (Fig 4.5)

(a) register – register operations

(b) register – immediate operations
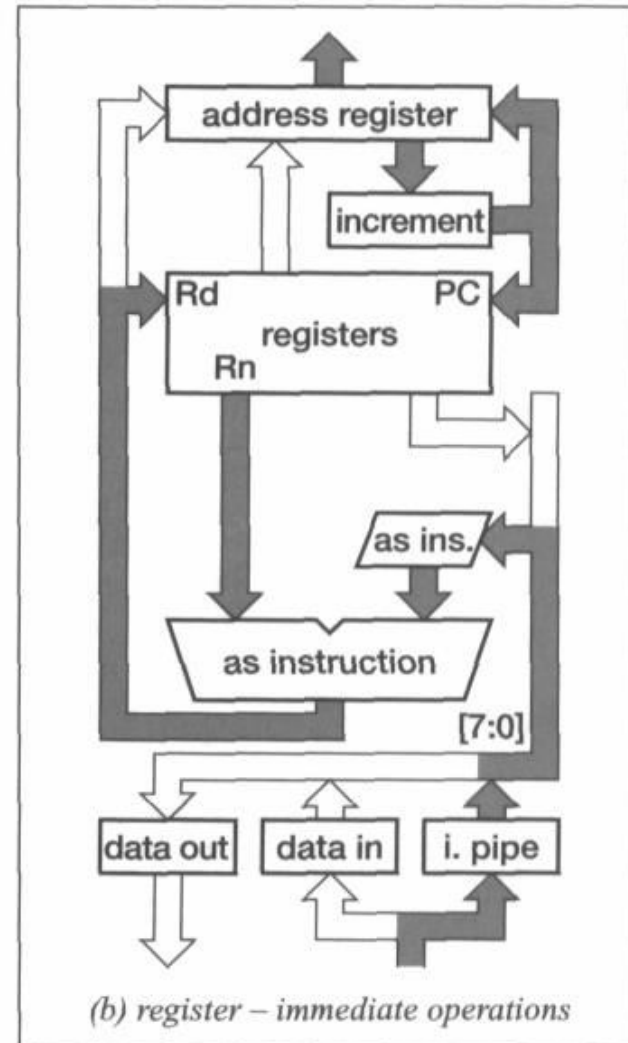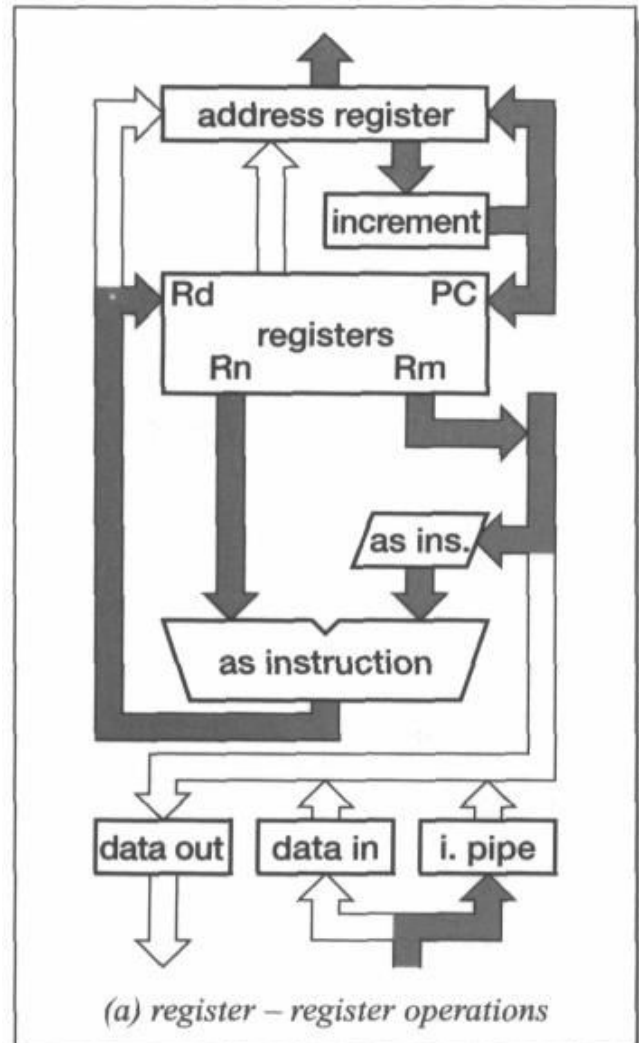
ARM instruction Execution

Data transfer instructions

- A data transfer (load or store) instruction computes a memory address in a manner very similar to the way a data processing instruction computes its result.

- A register is used as the base address, to which is added (or from which is subtracted) an offset which again may be another register or an immediate value.

- This time, however, a 12-bit immediate value is used without a shift operation rather than a shifted 8-bit value. The address is sent to the address register(IN 1ST CYCLE), and in a second cycle the data transfer takes place.

ARM instruction Execution

Data transfer instructions

When does the auto indexing takes place?

- Rather than leave the datapath largely idle during the data transfer cycle, the ALU holds the address components from the first cycle and is available to compute an auto-indexing modification to the base register if this is required.

- (If auto-indexing is not required the computed value is not written back to the base register in the second cycle.)

ARM instruction Execution

Figure 4.6 SIR (store register) datapath activity.

(a) 1st cycle – compute address

(b) 2nd cycle – store data & auto-index
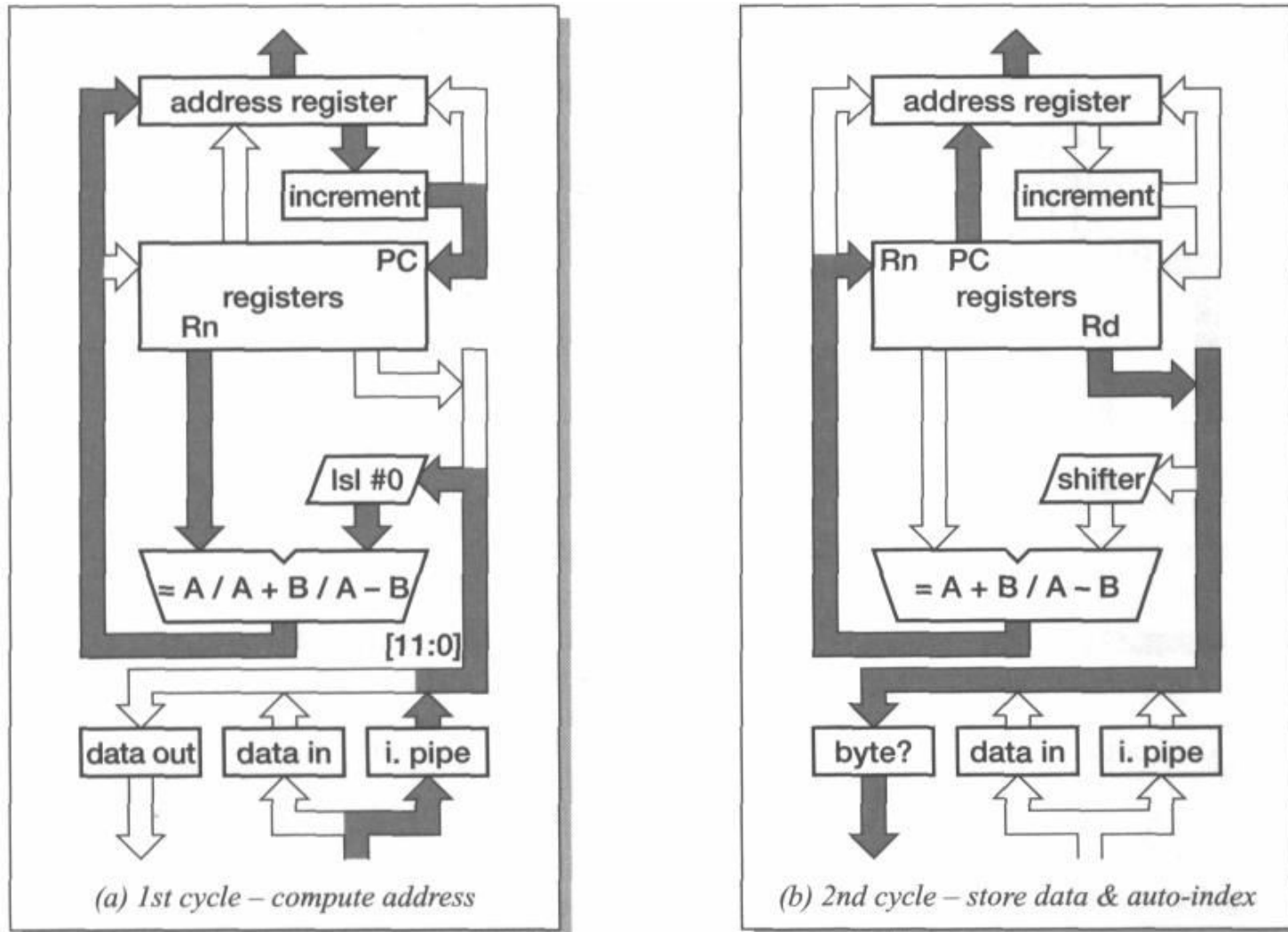
ARM instruction Execution

Data transfer instructions

Address register (Can it be called as the pipeline register)??

- The incremented PC value is stored in the register bank at the end of the first cycle so that the address register is free to accept the data transfer address for the second cycle, then at the end of the second cycle the PC is fed back to the address register to allow instruction prefetching to continue.

- At this stage the value sent to the address register in a cycle is the value used for the memory access in *the following* cycle.

- The address register is, in effect, a pipeline register between the processor datapath and the external memory.

ARM instruction Execution

Data transfer instructions

Byte date transfer:

- When the instruction specifies the store of a byte data type, the 'data out' block extracts the bottom byte from the register and replicates it four times across the 32-bit data bus.

- External memory control logic can then use the bottom two bits of the address bus to activate the appropriate byte within the memory system.

ARM instruction Execution

Data transfer instructions

How about Load?

- It takes 3 cycles.

- Load instructions follow a similar pattern like Store except that the data from memory only gets as far as the 'data in' register on the second cycle and a third cycle is needed to transfer the data from there to the destination register.

ARM instruction Execution

Branch instructions

- Branch instructions compute the target address in the first cycle.

  - A 24-bit immediate field is extracted from the instruction and then shifted left two bit positions to give a word-aligned offset which is added to the PC.

  - The result is issued as an instruction fetch address, and while the instruction pipeline refills the return address is copied into the link register (r14) if this is required (that is, if the instruction is a 'branch with link').

ARM instruction Execution (Branch)
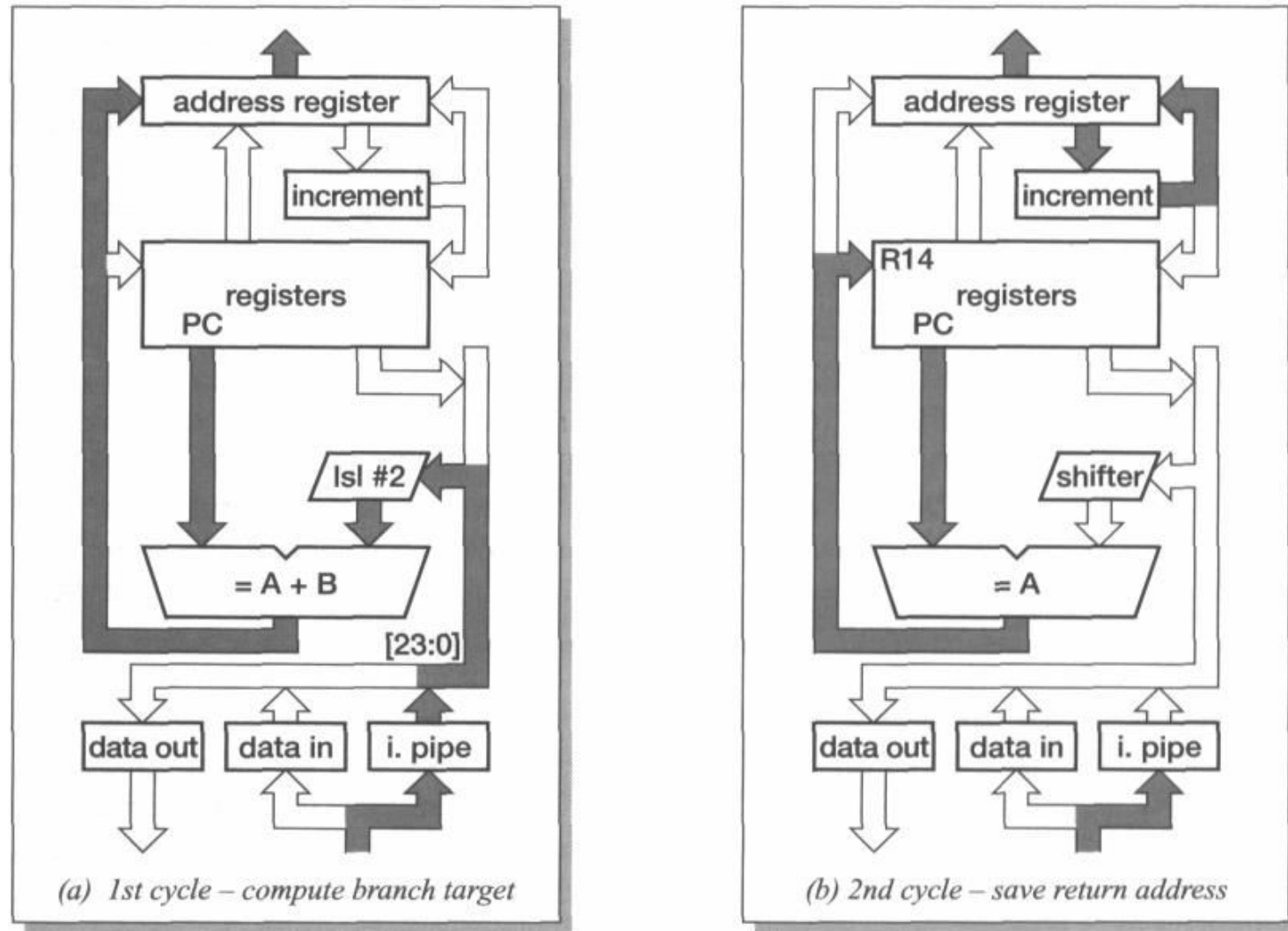
Figure 4.7 The first two (of three) cycles of a branch instruction

ARM instruction Execution

Branch instructions

- The third cycle, which is required to complete the pipeline refilling, is also used to make a small correction to the value stored in the link register in order that it points directly at the instruction which follows the branch.

- This is necessary because r15 contains pc + 8 whereas the address of the next instruction is pc + 4

Much awaited diagram

- ARM 5-stage pipeline Organization

Figure 4.4 ARM9TDMI 5-stage pipeline organization.
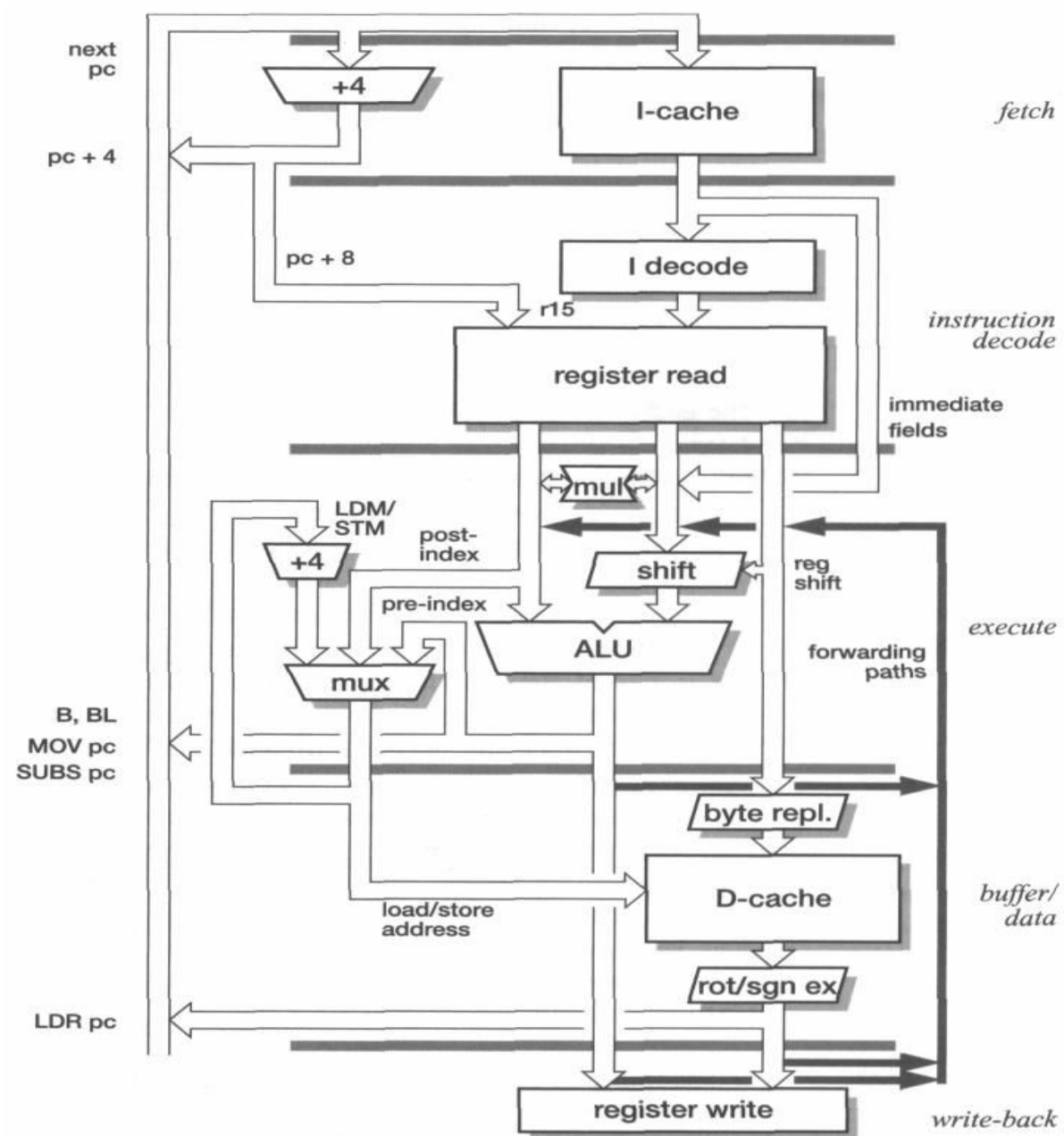
ARM 5-stage pipeline Organization

- Fetch
  - The instruction is fetched from memory and placed in the instruction pipeline.

  - The 5-stage pipeline reads the instruction operands one stage earlier in the pipeline, and would naturally get a different value (PC+4 rather than PC+8).

  - Referring to Figure 4.4, the incremented PC value from the fetch stage is fed directly to the register file in the decode stage, bypassing the pipeline register between the two stages. PC+4 for the next instruction is equal to PC+8 for the current instruction, so the correct r15 value is obtained without additional hardware.

ARM 5-stage pipeline Organization

- Decode
    - The instruction is decoded and register operands read from the register file.
    - There are three operand read ports in the register file, so most ARM instructions can source all their operands in one cycle.

ARM 5-stage pipeline Organization

- Execute
  - An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU.

ARM 5-stage pipeline Organization

- Buffer/Data (read)
  - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock cycle to give the same pipeline flow for all instructions.

- Data Forwarding:
  - A major source of complexity in the 5-stage pipeline (compared to the 3-stage pipeline) is that, because instruction execution is spread across three pipeline stages, the only way to resolve data dependencies without stalling the pipeline is to introduce forwarding paths.

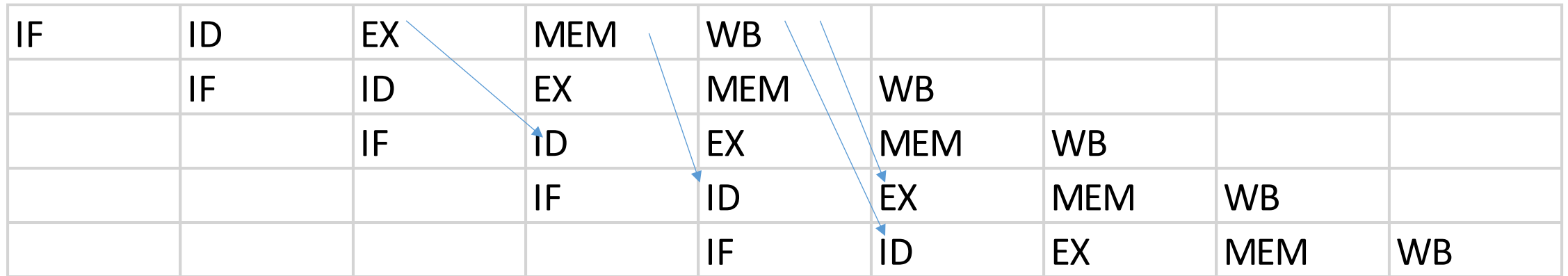# ARM 5-stage pipeline Organization

- Data Forwarding:
  - Observe the forwarding paths in the diagram (Fig 4.4)

  - Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM pipeline requires each of the three source operands to be forwarded from any of three intermediate result registers

ARM 5-stage pipeline Organization

- What are the three source Operands?
    1. BUS A
    2. BUS B
    3. PC

- What are the intermediate result registers?
    1. ALU Output
    2. Data available after store (in 4th cycle)
    3. Data available at the first half of 5th cycle (can be written in the first half of 5th CC and can be read in second half of 5th CC) (transparent register file – you have study in MIPS)

ARM 5-stage pipeline Organization

IF	ID	EX	MEM	WB				
	IF	ID	EX	MEM	WB			
		IF	ID	EX	MEM	WB		
			IF	ID	EX	MEM	WB	
				IF	ID	EX	MEM	WB

The forwarding Paths

ARM 5-stage pipeline Organization (Stalls)

- There is one case where, even with forwarding, it is not possible to avoid a pipeline stall.

- Consider the following code sequence:
  LDR rN, [. .] ;   load rN from somewhere
  ADD r2, r1, rN ;   and use it immediately

- The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage.

- The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

ARM 5-stage pipeline Organization (Stalls)

- There is one case where, even with forwarding, it is not possible to avoid a pipeline stall.

- Consider the following code sequence:
  LDR rN, [. .] ;   load rN from somewhere
  ADD r2, r1, rN ;   and use it immediately

- The processor cannot avoid a one-cycle stall as the value loaded into rN only enters the processor at the end of the buffer/data stage and it is needed by the following instruction at the start of the execute stage.

- The only way to avoid this stall is to encourage the compiler (or assembly language programmer) not to put a dependent instruction immediately after a load instruction.

ARM 5-stage pipeline Organization (Stalls)

- Consider the following code sequence:
  LDR rN, [. .] ;   load rN from somewhere
  ADD r2, r1, rN ;   and use it immediately

Load		IF	ID	EX	MEM	WB		
ADD			IF	ID	EX	MEM	WB	
				NOT POSSIBLE				
Load		IF	ID	EX	MEM	WB		
Stall			IF	ID	STALL	EX	MEM	WB
Load		IF	ID	EX	MEM	WB		
some inst			IF	ID	EX	MEM	WB	
ADD				IF	ID	EX	MEM	WB

ARM 5-stage pipeline Organization (Stalls)

- What is the role of MUX before the input to D-Cache

  - The MUX is for the address to be accessed.(see the figure)

Lecture 6

Previous Class

- 3 Stage Pipeline

- 5 Stage Pipeline

Today

- Instruction Set – Contd…

  - SWI

SOFTWARE INTERRUPT (SWI)

- The software interrupt instruction is used for calls to the operating system and is often called a 'supervisor call'.

- It puts the processor into supervisor mode and begins executing instructions from address 0x08.

SOFTWARE INTERRUPT (SWI)

- Binary encoding
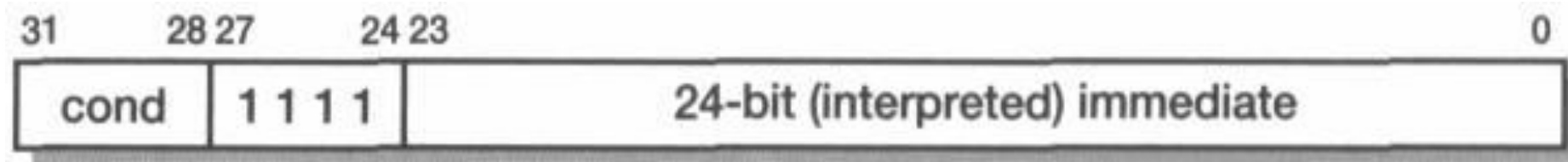
Figure 5.5 Software interrupt binary encoding.
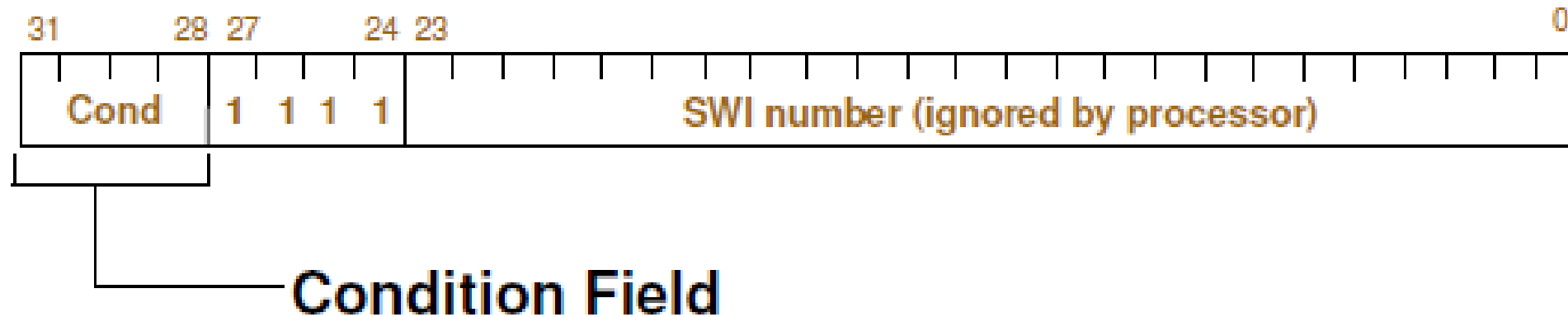
- **The 24-bit immediate field does not influence the operation of the instruction but may be interpreted by the system code.**

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.

SOFTWARE INTERRUPT (SWI)

- If the condition is passed the instruction enters supervisor mode using the standard ARM exception entry sequence.

- In detail, the processor actions are:

   1. Save the address of the instruction after the SWI in r14_svc.

   2. Save the CPSR in SPSR_svc.

   3. Enter supervisor mode and disable IRQs (but not FIQs) by setting CPSR[4:0] to $(10011)_2$ and CPSR[7] to 1.

   4. Set the PC to 08 and begin executing the instructions there.

SOFTWARE INTERRUPT (SWI)

- To return to the instruction after the SWI the system routine must not only copy r14_svc back into the PC, but it must also restore the CPSR from SPSR_svc.

- This requires the use of one of the special forms of the data processing instruction.

SOFTWARE INTERRUPT (SWI)

- To return to the instruction after the SWI the system routine must not only copy r14_svc back into the PC, but it must also restore the CPSR from SPSR_svc.

- This requires the use of one of the special forms of the data processing instruction.

SOFTWARE INTERRUPT (SWI)

- Syntax:

- **SWI{<cond>} <SWI number>**

PSR TRANSFER INSTRUCTIONS

- MRS allow contents of CPSR/SPSR to be transferred from appropriate status register to a general purpose register.

  - All of status register, or just the flags, can be transferred.

- Syntax:

  - MRS{<cond>} Rd,<psr>   ; Rd = <psr>

  - Where <psr> = CPSR, CPSR_all, SPSR or SPSR_all

  Examples: MRS r0, CPSR ; move the CPSR to r0 ;

                 MRS r3, SPSR ;move the SPSR to r3

PSR TRANSFER INSTRUCTIONS

- MRS (immediate form)

  - MSR{<cond>} <psrf>,#Immediate

  - Where <psrf> = CPSR_flg or SPSR_flg

  - This immediate must be a 32-bit immediate, of which the 4 most significant bits are written to the flag bits.

PSR TRANSFER INSTRUCTIONS

- MSR allow contents of a general purpose register   to be transferred to CPSR/SPSR.

  - All of status register, or just the flags, can be transferred.

- Syntax:

  - MSR{<cond>} <psr>,Rm   ; <psr> = Rm

  - MSR{<cond>} <psrf>,Rm   ; <psrf> = Rm

    Where <psr> = CPSR, CPSR_all, SPSR or SPSR_all

    <psrf> = CPSR_flg or SPSR_flg

USING MRS AND MSR

- Currently reserved bits, may be used in future, therefore:
    - they must be preserved when altering PSR
    - the value they return must not be relied upon when testing other bits.
- Thus read-modify-write strategy must be followed when modifying any PSR:
    - Transfer PSR to register using MRS
    - Modify relevant bits
    - Transfer updated value back to PSR using MSR
- Note:
    - In User Mode, all bits can be read but only the flag bits can be written to.
    - The SPSR form should not be used in user or system mode since there is no accessible SPSR in those modes.

Lecture 7

Previous Class

- SWI

Today

- Instruction Set – Contd…

   Branch

BRANCH INSTRUCTIONS

- Branch and Branch with Link (B, BL)
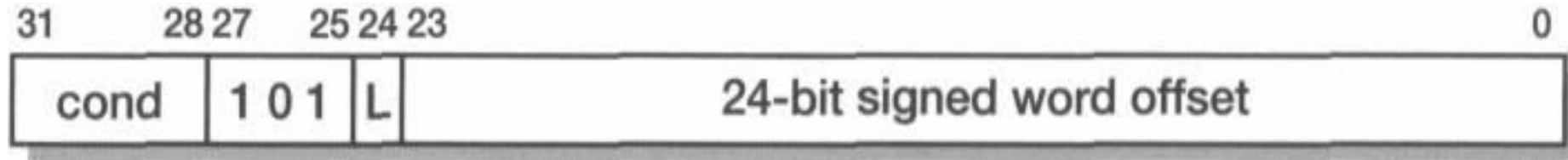
Figure 5.3 Branch and Branch with Link binary encoding.

- Branch and Branch with Link instructions cause the processor to begin executing instructions from an address computed by sign extending the 24-bit offset specified in the instruction, shifting it left two places to form a word offset, then adding it to the program counter which contains the address of the branch instruction plus eight bytes.

The assembler will compute the correct offset under normal circumstances.

The range of the branch instruction is +/- 32 Mbytes.

BRANCH INSTRUCTIONS

- Branch and Branch with Link (B, BL)

31		28 27		25 24 23			0
cond		1 0 1	L		24-bit signed word offset		
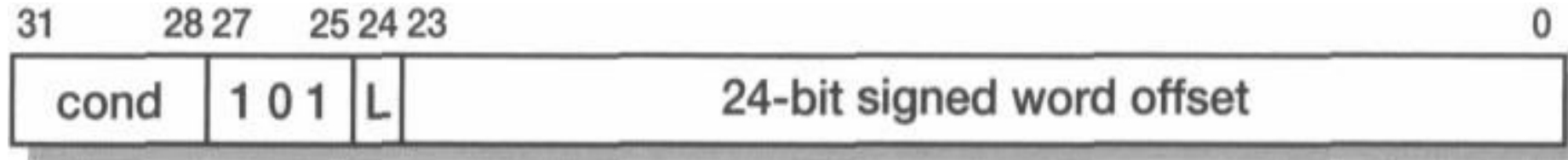
Figure 5.3 Branch and Branch with Link binary encoding.

- The Branch with Link variant, which has the L bit (bit 24) set, also moves the address of the instruction following the branch into the link register (r14) of the current processor mode. This is normally used to perform a subroutine call, with the return being caused by copying the link register back into the PC.

- Both forms of the instruction may be executed conditionally or unconditionally.

BRANCH INSTRUCTIONS

- Both forms of the instruction may be executed conditionally or unconditionally.

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

  '<target address>' or subroutine_label is normally a label in the assembler code; the assembler will generate the offset (which will be the difference between the address of the target and the address of the branch instruction plus 8).

ARM BRANCHES AND SUBROUTINES

- B \<label>
  - PC relative. ±32 Mbyte range.

- BL \<subroutine>
  - Stores return address in LR
  - Returning implemented by restoring the PC from LR
  - For non-leaf functions, LR will have to be stacked

func1

func2

```
BL func1
```

```
STMFD
sp!,{regs,lr}
    :
  BL  func2
    :
  LDMFD
sp!,{regs,pc}
```

```
MOV pc, lr
```
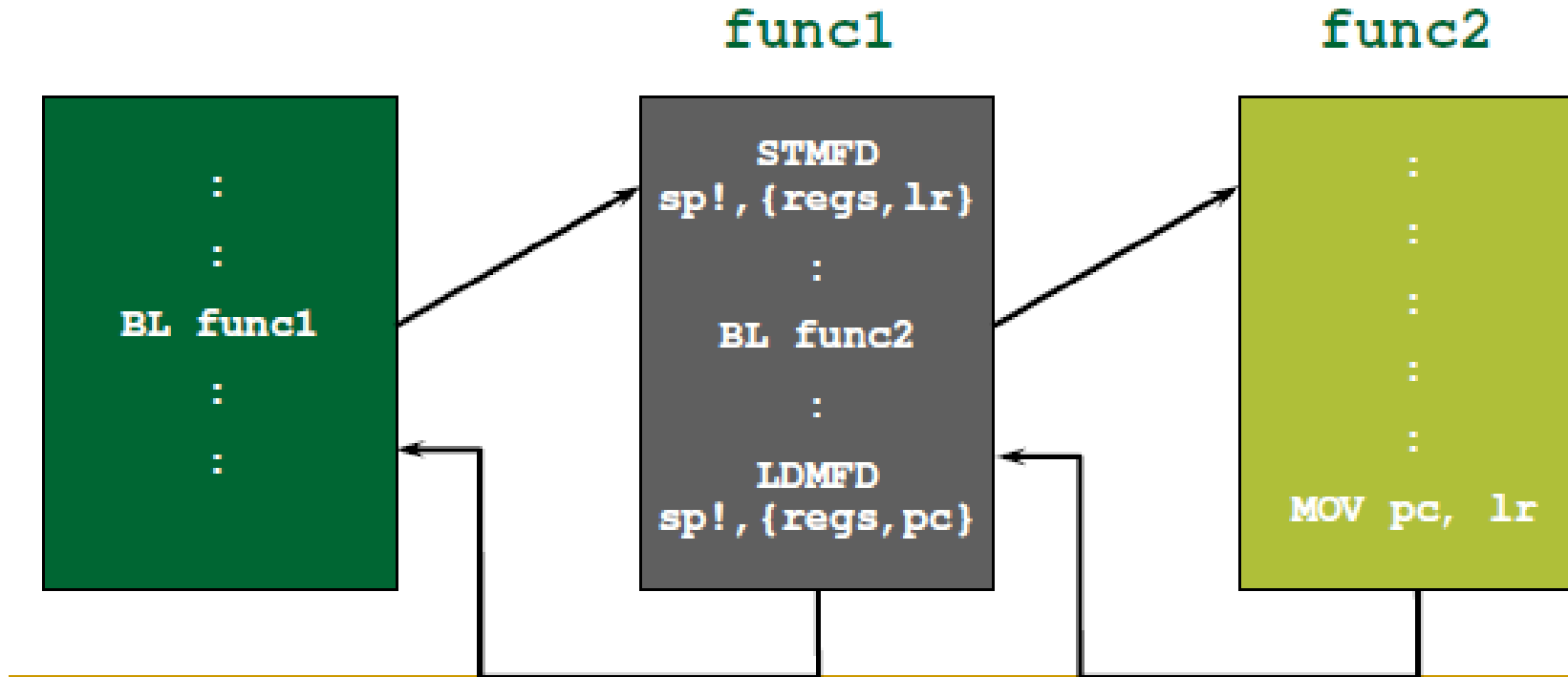
ARM SUBROUTINE LINKAGE

- Branch and link instruction:

    BL foo

  - Copies current PC to r14.

- To return from subroutine:

  - MOV r15,r14

Examples

- An unconditional jump:

```
                              ; unconditional jump ..
         ..
LABEL    ..                   ; .. to here
```

Examples

To execute a loop ten times:

```
            MOV     r0, #10 ; initialize loop counter
LOOP        ..
            SUBS    r0, #1  ; decrement counter setting CCs
            BNE     LOOP    ; if counter <> 0 repeat loop..
            ..              ; .. else drop through
```

Examples

To call a subroutine:

```
            ..
        BL        SUB        ; branch and link to subroutine SUB
            ..                ; return to here
            ..
SUB         ..                ; subroutine entry point
        MOV       PC, r14 ; return
```

Examples

Conditional subroutine call:

```
..
CMP        r0, #5    ; if r0 < 5
BLLT       SUB1      ; then call SUB1
BLGE       SUB2      ; else call SUB2
..
```

- This example will only work correctly if SUB1 does not change the condition codes, since if the BLLT is taken it will return to the BLGE.

    If the condition codes are changed by SUB1, SUB2 may be executed as well.

Branches

Note: Branches which attempt to go past the beginning or the end of the 32-bit address space should be avoided since they may have unpredictable results.

Conditional branches (branch conditions)

The pairs of conditions which are listed in the same row of the table (for instance BCC and BLO) are synonyms which result in identical binary code, but both are available because each makes the interpretation of the assembly source code easier in particular circumstances.

Branch	Interpretation	Normal uses
B/BAL	Unconditional Always	Always take this branch 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 BLO	Carry clear Lower	Arithmetic operation did not give carry-out Unsigned comparison gave lower
BCS BHS	Carry set Higher or same	Arithmetic operation gave carry-out 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

ARM ADR PSEUDO-OP

- Cannot refer to an address directly in an instruction.

- Generate value by performing arithmetic on PC.

- ADR pseudo-op generates instruction required to calculate address:

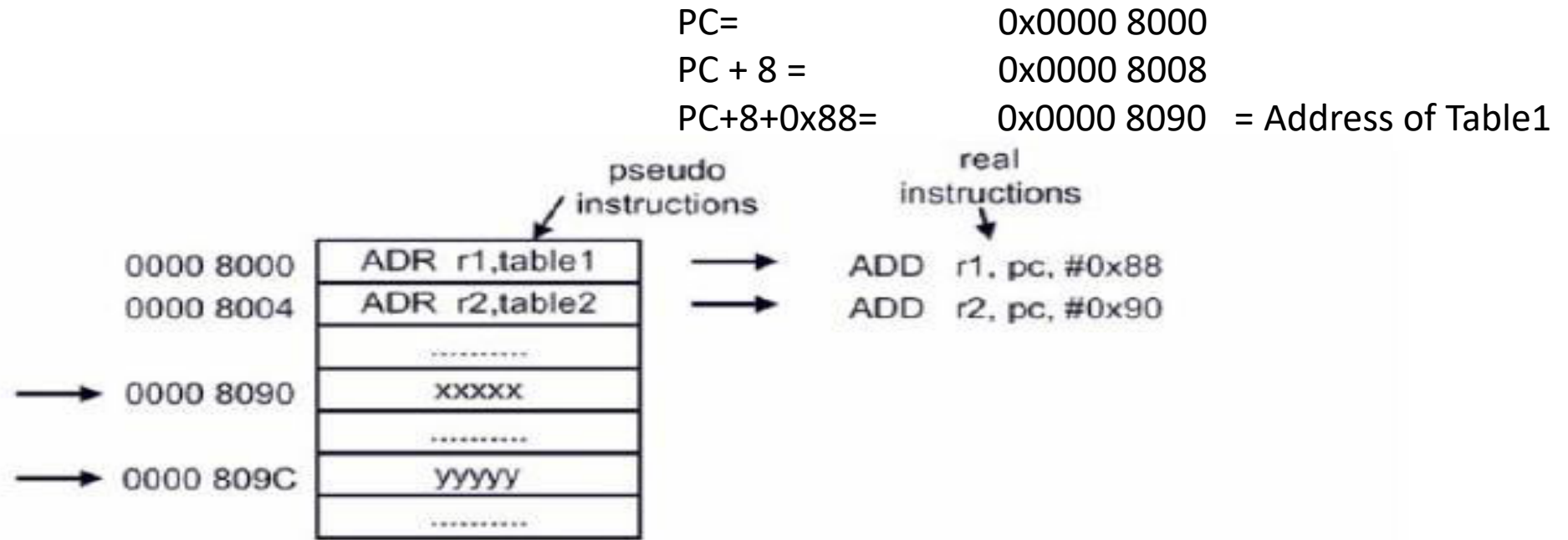 ADR r1,FOO

CONTD...

```
COPY:   ADR r1, TABLE1  ; r1 points to TABLE1
        ADR r2, TABLE2  ; r2 points to TABLE2
LOOP:   LDR r0, [r1]
        STR r0, [r2]
        ADD r1, r1, #4
        ADD r2, r2, #4
        ...
TABLE1:         ...
TABLE2:...
```

- Need to initialize address in r1 in the first place. How?
- Use ADR pseudo instruction - looks like normal instruction, but it is actually an assembler directive.
- The assembler translates it to one or more real instructions.

```
COPY:   ADR r1, TABLE1  ; r1 points to TABLE1
        ADR r2, TABLE2  ; r2 points to TABLE2
LOOP:   LDR r0, [r1], #4
        STR r0, [r2], #4
        ...
TABLE1:         ...
```

HOW ADR WORKS?

PC=   0x0000 8000
PC + 8 =   0x0000 8008
PC+8+0x88=   0x0000 8090   = Address of Table1

```
                              pseudo                      real
                            / instructions             instructions
                                  ↓                         ↓
0000 8000   |  ADR  r1,table1  |  ──────→     ADD   r1, pc, #0x88
0000 8004   |  ADR  r2,table2  |  ──────→     ADD   r2, pc, #0x90
            |  ............    |
→ 0000 8090 |  xxxxx           |
            |  ............    |
→ 0000 809C |  yyyyy           |
            |  ............    |
```

- How does the **ADR** directive work? Address is 32-bit, difficult to put a 32-bit address value in a register in the first place
- Solution: Program Counter PC (**r15**) is often close to the desired data address value
- **ADR r1, TABLE1** is translated into an instruction that adds or subtracts a constant to PC (**r15**), and puts the result in r1
- This constant is known as **PC-relative offset**, and it is calculated as: addr_of_table1 - (PC_value + 8)

SUMMARY

- Load/store architecture
- Most instructions are RISCy, operate in single cycle.
- Some multi-register operations take longer.
- All instructions can be executed conditionally.