Table 6.3 Condition mnemonics

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Other data-processing instructions will set the condition flags when the instruction mnemonic is followed by "S." For example, `SUBS R2, R3, R7` will subtract R7 from R3, put the result in R2, and set the condition flags. Table B.5 in Appendix B summarizes which condition flags are influenced by each instruction. All data-processing instructions will affect the *N* and *Z* flags based on whether the result is zero or has the most significant bit set. `ADDS` and `SUBS` also influence *V* and *C*, and shifts influence *C*.

Code Example 6.10 shows instructions that execute conditionally. The first instruction, `CMP R2, R3`, executes unconditionally and sets the

Condition mnemonics differ for signed and unsigned comparison. For example, ARM provides two forms of greater than or equal comparison: HS (CS) is used for unsigned numbers and GE for signed. For unsigned numbers, $A - B$ will produce a carry out (C) when $A \geq B$. For signed numbers, $A - B$ will make N and V either both 0 or both 1 when $A \geq B$. Figure 6.7 highlights the difference between HS and GE comparisons with two examples using 4-bit numbers for ease of interpretation.

	Unsigned	Signed
$A = 1001_2$	A = 9	A = –7
$B = 0010_2$	B = 2	B = 2

A – B: 1001 + 1110 = 10111 (a)

$NZCV = 0011_2$

HS: TRUE

GE: FALSE

	Unsigned	Signed
$A = 0101_2$	A = 5	A = 5
$B = 1101_2$	B = 13	B = –3

A – B: 0101 + 0011 = 1000 (b)

$NZCV = 1001_2$

HS: FALSE

GE: TRUE

Figure 6.7 Signed vs. unsigned comparison: HS vs. GE

Code Example 6.16 WHILE LOOP

High-Level Code

```
int pow = 1;
int x = 0;

while (pow != 128) {
  pow = pow * 2;
  x = x + 1;
}
```

ARM Assembly Code

```
; R0 = pow, R1 = x
  MOV R0, #1      ; pow = 1
  MOV R1, #0      ; x = 0

WHILE
  CMP R0, #128    ; pow != 128 ?
  BEQ DONE        ; if pow == 128, exit loop
  LSL R0, R0, #1  ; pow = pow * 2
  ADD R1, R1, #1  ; x = x + 1
  B   WHILE       ; repeat loop
DONE
```

Code Example 6.17 FOR LOOP

High-Level Code

```
int i;
int sum = 0;


for (i = 0; i < 10; i = i + 1) {
  sum = sum + i;
}
```

ARM Assembly Code

```
; R0 = i, R1 = sum
  MOV R1, #0        ; sum = 0
  MOV R0, #0        ; i = 0          loop initialization

FOR
  CMP R0, #10       ; i < 10 ?       check condition
  BGE DONE          ; if (i >= 10) exit loop
  ADD R1, R1, R0    ; sum = sum + i  loop body
  ADD R0, R0, #1    ; i = i + 1      loop operation
  B   FOR           ; repeat loop
DONE
```

Code Example 6.18 ACCESSING ARRAYS USING A FOR LOOP

High-Level Code

```
int i;
int scores[200];
...


for (i = 0; i < 200; i = i + 1)


  scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address, R1 = i
; initialization code        ...
  MOV R0, #0x14000000        ; R0 = base address
  MOV R1, #0                 ; i = 0

LOOP
  CMP R1, #200               ; i < 200?
  BGE L3                     ; if i ≥ 200, exit loop
  LSL R2, R1, #2             ; R2 = i * 4
  LDR R3, [R0, R2]           ; R3 = scores[i]
  ADD R3, R3, #10            ; R3 = scores[i] + 10
  STR R3, [R0, R2]           ; scores[i] = scores[i] + 10
  ADD R1, R1, #1             ; i = i + 1
  B   LOOP                   ; repeat loop
L3
```

Code Example 6.19 FOR LOOP USING POST-INDEXING

High-Level Code

```
int i;
int scores[200];
...

for (i = 0; i < 200; i = i + 1)
        scores[i] = scores[i] + 10;
```

ARM Assembly Code

```
; R0 = array base address
; initialization code      ...
  MOV R0, #0x14000000      ; R0 = base address
  ADD R1, R0, #800         ; R1 = base address + (200*4)

LOOP
  CMP R0, R1               ; reached end of array?
  BGE L3                   ; if yes, exit loop
  LDR R2, [R0]             ; R2 = scores[i]
  ADD R2, R2, #10          ; R2 = scores[i] + 10
  STR R2, [R0], #4         ; scores[i] = scores[i] + 10
                           ; then R0 = R0 + 4
  B   LOOP                 ; repeat loop
  L3
```

Code Example 6.22 FUNCTION SAVING REGISTERS ON THE STACK

ARM Assembly Code

```
;R4 = result
DIFFOFSUMS
  SUB   SP, SP, #12       ; make space on stack for 3 registers
  STR   R9, [SP, #8]      ; save R9 on stack
  STR   R8, [SP, #4]      ; save R8 on stack
  STR   R4, [SP]          ; save R4 on stack

  ADD   R8, R0, R1        ; R8 = f + g
  ADD   R9, R2, R3        ; R9 = h + i
  SUB   R4, R8, R9        ; result = (f + g) - (h + i)
  MOV   R0, R4            ; put return value in R0

  LDR   R4, [SP]          ; restore R4 from stack
  LDR   R8, [SP, #4]      ; restore R8 from stack
  LDR   R9, [SP, #8]      ; restore R9 from stack
  ADD   SP, SP, #12       ; deallocate stack space

  MOV   PC, LR            ; return to caller
```

Code Example 6.23 SAVING AND RESTORING MULTIPLE REGISTERS

ARM Assembly Code

```
; R4 = result
DIFFOFSUMS
  STMFD  SP!, {R4, R8, R9}        ; push R4/8/9 on full descending stack

  ADD    R8, R0, R1               ; R8 = f + g
  ADD    R9, R2, R3               ; R9 = h + i
  SUB    R4, R8, R9               ; result = (f + g) - (h + i)
  MOV    R0, R4                   ; put return value in R0

  LDMFD  SP!, {R4, R8, R9}        ; pop R4/8/9 off full descending stack

  MOV    PC, LR                   ; return to caller
```

Code Example 6.26 NONLEAF FUNCTION CALL

High-Level Code

```
int f1(int a, int b) {
  int i, x;

  x = (a + b)*(a - b);
  for (i=0; i<a; i++)
    x = x + f2(b+i);
  return x;
}


int f2(int p) {
  int r;

  r = p + 5;
  return r + p;
}
```

ARM Assembly Code

```
; R0 = a, R1 = b, R4 = i, R5 = x
F1
   PUSH  {R4, R5, LR}    ; save preserved registers used by f1
   ADD   R5, R0, R1      ; x = (a + b)
   SUB   R12, R0, R1     ; temp = (a - b)
   MUL   R5, R5, R12     ; x = x * temp = (a + b) * (a - b)
   MOV   R4, #0          ; i = 0
FOR
   CMP   R4, R0          ; i < a?
   BGE   RETURN          ; no: exit loop
   PUSH  {R0, R1}        ; save nonpreserved registers
   ADD   R0, R1, R4      ; argument is b + i
   BL    F2              ; call f2(b+i)
   ADD   R5, R5, R0      ; x = x + f2(b+i)
   POP   {R0, R1}        ; restore nonpreserved registers
   ADD   R4, R4, #1      ; i++
   B     FOR             ; continue for loop
RETURN
   MOV   R0, R5          ; return value is x
   POP   {R4, R5, LR}    ; restore preserved registers
   MOV   PC, LR          ; return from f1

; R0 = p, R4 = r
F2
   PUSH  {R4}            ; save preserved registers used by f2
   ADD   R4, R0, 5       ; r = p + 5
   ADD   R0, R4, R0      ; return value is r + p
   POP   {R4}            ; restore preserved registers
   MOV   PC, LR          ; return from f2
```

On careful inspection, one might note that f2 does not modify R1, so f1 did not need to save and restore it. However, a compiler cannot always easily ascertain which nonpreserved registers may be disturbed during a function call. Hence, a simple compiler will always make the caller save and restore any nonpreserved registers that it needs after the call.

An optimizing compiler could observe that f2 is a leaf procedure and could allocate r to a nonpreserved register, avoiding the need to save and restore R4.

Code Example 6.27 factorial RECURSIVE FUNCTION CALL

High-Level Code

```
int factorial(int n) {
  if (n <= 1)
    return 1;



  else
    return (n * factorial(n - 1));
}
```

ARM Assembly Code

```
0x8500  FACTORIAL  PUSH  {R0, LR}    ; push n and LR on stack
0x8504             CMP   R0, #1      ; R0 <= 1?
0x8508             BGT   ELSE        ; no: branch to else
0x850C             MOV   R0, #1      ; otherwise, return 1
0x8510             ADD   SP, SP, #8  ; restore SP
0x8514             MOV   PC, LR      ; return
0x8518  ELSE       SUB   R0, R0, #1  ; n = n - 1
0x851C             BL    FACTORIAL   ; recursive call
0x8520             POP   {R1, LR}    ; pop n (into R1) and LR
0x8524             MUL   R0, R1, R0  ; R0 = n * factorial(n - 1)
0x8528             MOV   PC, LR      ; return
```

Assembly Code	Field Values											Machine Code										
	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0	31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
`LSL R0, R9, #7` (0xE1A00389)	1110_2	00_2	0	1101_2	0	0	0	7	00_2	0	9	1110	00	0	1101	0	0000	0000	00111	00	0	1001
`ROR R3, R5, #21` (0xE1A03AE5)	1110_2	00_2	0	1101_2	0	0	3	21	11_2	0	5	1110	00	0	1101	0	0000	0011	10101	11	0	0101
	cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm	cond	op	I	cmd	S	Rn	Rd	shamt5	sh		Rm

Figure 6.20 Shift instructions with immediate shift amounts

Assembly Code	Field Values												Machine Code											
	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0	31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
`LSR R4, R8, R6` (0xE1A04638)	1110_2	00_2	0	1101_2	0	0	4	6	0	01_2	1	8	1110	00	0	1101	0	0000	0100	0110	0	01	1	1000
`ASR R5, R1, R12` (0xE1A05C51)	1110_2	00_2	0	1101_2	0	0	5	12	0	10_2	1	1	1110	00	0	1101	0	0000	0101	1100	0	10	1	0001
	cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm	cond	op	I	cmd	S	Rn	Rd	Rs		sh		Rm

Figure 6.21 Shift instructions with register shift amounts

Table 6.12 ARM operand addressing modes

Operand Addressing Mode	Example	Description
Register		
Register-only	ADD R3, R2, R1	R3 ← R2 + R1
Immediate-shifted register	SUB R4, R5, R9, LSR #2	R4 ← R5 – (R9 >> 2)
Register-shifted register	ORR R0, R10, R2, ROR R7	R0 ← R10 | (R2 ROR R7)
Immediate	SUB R3, R2, #25	R3 ← R2 – 25
Base		
Immediate offset	STR R6, [R11, #77]	mem[R11+77] ← R6
Register offset	LDR R12, [R1, –R5]	R12 ← mem[R1 – R5]
Immediate-shifted register offset	LDR R8, [R9, R2, LSL #2]	R8 ← mem[R9 + (R2 << 2)]
PC-Relative	B LABEL1	Branch to LABEL1

```
module arm(input  logic          clk, reset,
           output logic [31:0]   PC,
           input  logic [31:0]   Instr,
           output logic          MemWrite,
           output logic [31:0]   ALUResult, WriteData,
           input  logic [31:0]   ReadData);

  logic [3:0] ALUFlags;
  logic       RegWrite,
              ALUSrc, MemtoReg, PCSrc;
  logic [1:0] RegSrc, ImmSrc, ALUControl;

  controller c(clk, reset, Instr[31:12], ALUFlags,
               RegSrc, RegWrite, ImmSrc,
               ALUSrc, ALUControl,
               MemWrite, MemtoReg, PCSrc);
  datapath dp(clk, reset,
              RegSrc, RegWrite, ImmSrc,
              ALUSrc, ALUControl,
              MemtoReg, PCSrc,
              ALUFlags, PC, Instr,
              ALUResult, WriteData, ReadData);
endmodule
```

SystemVerilog

```
module controller(input  logic          clk, reset,
                  input  logic [31:12]  Instr,
                  input  logic [3:0]    ALUFlags,
                  output logic [1:0]    RegSrc,
                  output logic          RegWrite,
                  output logic [1:0]    ImmSrc,
                  output logic          ALUSrc,
                  output logic [1:0]    ALUControl,
                  output logic          MemWrite, MemtoReg,
                  output logic          PCSrc);

  logic [1:0] FlagW;
  logic       PCS, RegW, MemW;

  decoder dec(Instr[27:26], Instr[25:20], Instr[15:12],
              FlagW, PCS, RegW, MemW,
              MemtoReg, ALUSrc, ImmSrc, RegSrc, ALUControl);
  condlogic cl(clk, reset, Instr[31:28], ALUFlags,
               FlagW, PCS, RegW, MemW,
               PCSrc, RegWrite, MemWrite);

endmodule
```

SystemVerilog

```
module decoder(input  logic [1:0] Op,
               input  logic [5:0] Funct,
               input  logic [3:0] Rd,
               output logic [1:0] FlagW,
               output logic       PCS, RegW, MemW,
               output logic       MemtoReg, ALUSrc,
               output logic [1:0] ImmSrc, RegSrc, ALUControl);

  logic [9:0] controls;
  logic       Branch, ALUOp;

  // Main Decoder
  always_comb
      casex(Op)
                                 // Data-processing immediate
        2'b00: if (Funct[5])     controls = 10'b0000101001;
                                 // Data-processing register
               else              controls = 10'b0000001001;
                                 // LDR
        2'b01: if (Funct[0])     controls = 10'b0001111000;
                                 // STR
               else              controls = 10'b1001110100;
                                 // B
        2'b10:                   controls = 10'b0110100010;
                                 // Unimplemented
        default:                 controls = 10'bx;
      endcase

  assign {RegSrc, ImmSrc, ALUSrc, MemtoReg,
    RegW, MemW, Branch, ALUOp} = controls;

  // ALU Decoder
  always_comb
  if (ALUOp) begin          // which DP Instr?
    case(Funct[4:1])
        4'b0100:  ALUControl = 2'b00;  // ADD
        4'b0010:  ALUControl = 2'b01;  // SUB
        4'b0000:  ALUControl = 2'b10;  // AND
        4'b1100:  ALUControl = 2'b11;  // ORR
        default:  ALUControl = 2'bx;   // unimplemented
    endcase

    // update flags if S bit is set (C & V only for arith)
    FlagW[1]      = Funct[0];
    FlagW[0]      = Funct[0] &
      (ALUControl == 2'b00 | ALUControl == 2'b01);
  end else begin
    ALUControl = 2'b00;  // add for non-DP instructions
    FlagW      = 2'b00;  // don't update Flags
  end

  // PC Logic
  assign PCS  = ((Rd == 4'b1111) & RegW) | Branch;
endmodule
```

SystemVerilog

```
module condlogic(input   logic         clk, reset,
                 input   logic [3:0]   Cond,
                 input   logic [3:0]   ALUFlags,
                 input   logic [1:0]   FlagW,
                 input   logic         PCS, RegW, MemW,
                 output  logic         PCSrc, RegWrite,
                                       MemWrite);

  logic [1:0] FlagWrite;
  logic [3:0] Flags;
  logic       CondEx;

  flopenr #(2)flagreg1(clk, reset, FlagWrite[1],
                       ALUFlags[3:2], Flags[3:2]);
  flopenr #(2)flagreg0(clk, reset, FlagWrite[0],
                       ALUFlags[1:0], Flags[1:0]);

  // write controls are conditional
  condcheck cc(Cond, Flags, CondEx);
  assign FlagWrite = FlagW & {2{CondEx}};
  assign RegWrite  = RegW  & CondEx;
  assign MemWrite  = MemW  & CondEx;
  assign PCSrc     = PCS   & CondEx;
endmodule

module condcheck(input  logic [3:0] Cond,
                 input  logic [3:0] Flags,
                 output logic       CondEx);

  logic neg, zero, carry, overflow, ge;

  assign {neg, zero, carry, overflow} = Flags;
  assign ge = (neg == overflow);

  always_comb
    case(Cond)
      4'b0000: CondEx = zero;             // EQ
      4'b0001: CondEx = ~zero;            // NE
      4'b0010: CondEx = carry;            // CS
      4'b0011: CondEx = ~carry;           // CC
      4'b0100: CondEx = neg;              // MI
      4'b0101: CondEx = ~neg;             // PL
      4'b0110: CondEx = overflow;         // VS
      4'b0111: CondEx = ~overflow;        // VC
      4'b1000: CondEx = carry & ~zero;    // HI
      4'b1001: CondEx = ~(carry & ~zero); // LS
      4'b1010: CondEx = ge;               // GE
      4'b1011: CondEx = ~ge;              // LT
      4'b1100: CondEx = ~zero & ge;       // GT
      4'b1101: CondEx = ~(~zero & ge);    // LE
      4'b1110: CondEx = 1'b1;             // Always
      default: CondEx = 1'bx;             // undefined
    endcase
endmodule
```

SystemVerilog

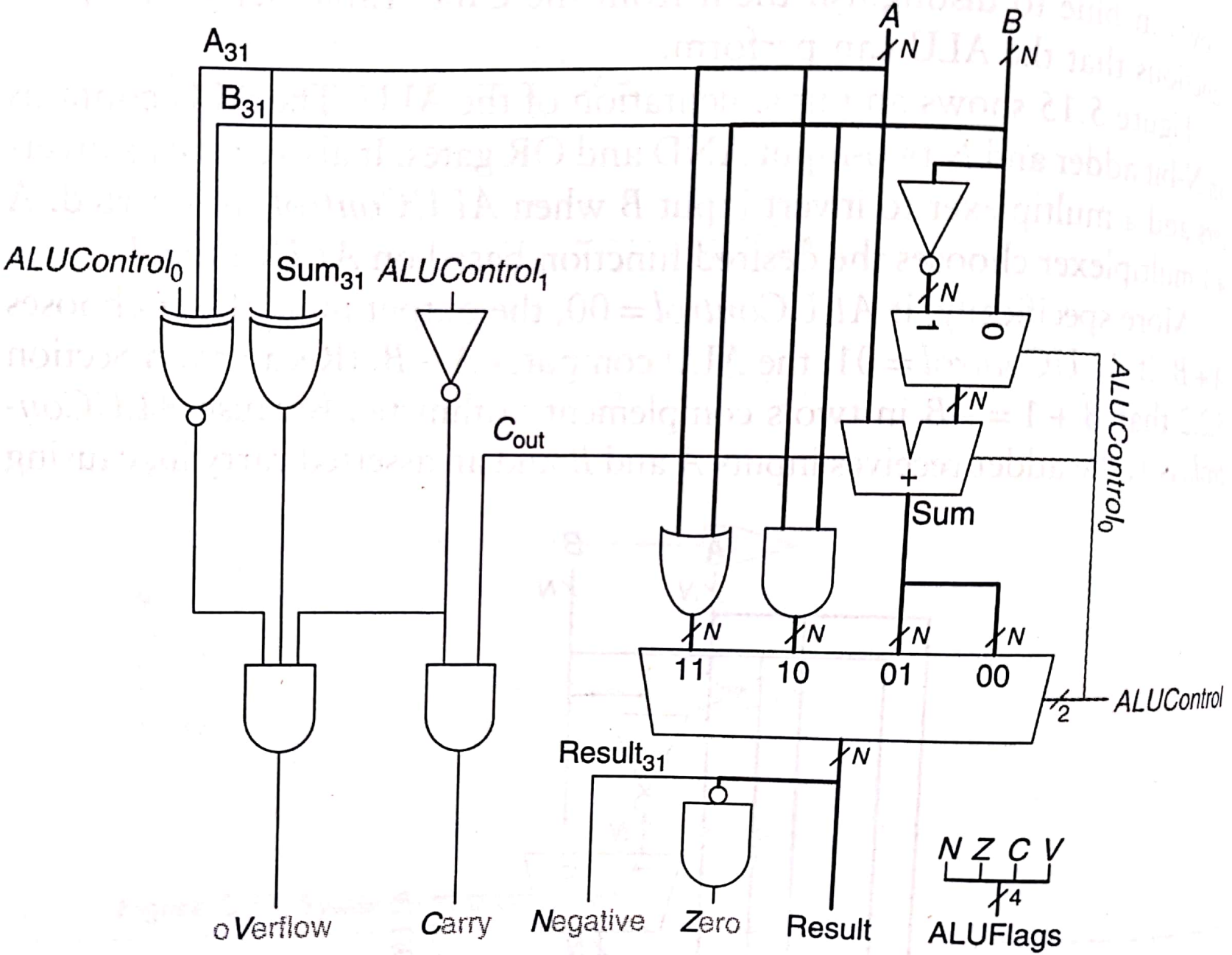
```
module datapath(input  logic         clk, reset,
                input  logic [1:0]   RegSrc,
                input  logic         RegWrite,
                input  logic [1:0]   ImmSrc,
                input  logic         ALUSrc,
                input  logic [1:0]   ALUControl,
                input  logic         MemtoReg,
                input  logic         PCSrc,
                output logic [3:0]   ALUFlags,
                output logic [31:0]  PC,
                input  logic [31:0]  Instr,
                output logic [31:0]  ALUResult, WriteData,
                input  logic [31:0]  ReadData);

  logic [31:0] PCNext, PCPlus4, PCPlus8;
  logic [31:0] ExtImm, SrcA, SrcB, Result;
  logic [3:0]  RA1, RA2;

  // next PC logic
  mux2 #(32)   pcmux(PCPlus4, Result, PCSrc, PCNext);
  flopr #(32)  pcreg(clk, reset, PCNext, PC);
  adder #(32)  pcadd1(PC, 32'b100, PCPlus4);
  adder #(32)  pcadd2(PCPlus4, 32'b100, PCPlus8);

  // register file logic
  mux2 #(4)  ra1mux(Instr[19:16], 4'b1111, RegSrc[0], RA1);
  mux2 #(4)  ra2mux(Instr[3:0], Instr[15:12], RegSrc[1], RA2);
  regfile    rf(clk, RegWrite, RA1, RA2,
                Instr[15:12], Result, PCPlus8,
                SrcA, WriteData);
  mux2 #(32) resmux(ALUResult, ReadData, MemtoReg, Result);
  extend     ext(Instr[23:0], ImmSrc, ExtImm);

  // ALU logic
  mux2 #(32) srcbmux(WriteData, ExtImm, ALUSrc, SrcB);
  alu        alu(SrcA, SrcB, ALUControl, ALUResult, ALUFlags);
endmodule
```


A
B
N
N
A_{31}
B_{31}
$ALUControl_0$
Sum_{31}
$ALUControl_1$
N
1
0
N
C_{out}
+
Sum
$ALUControl_0$
N
N
N
N
11
10
01
00
2
ALUControl
$Result_{31}$
N
N Z C V
4
oVerflow
Carry
Negative
Zero
Result
ALUFlags

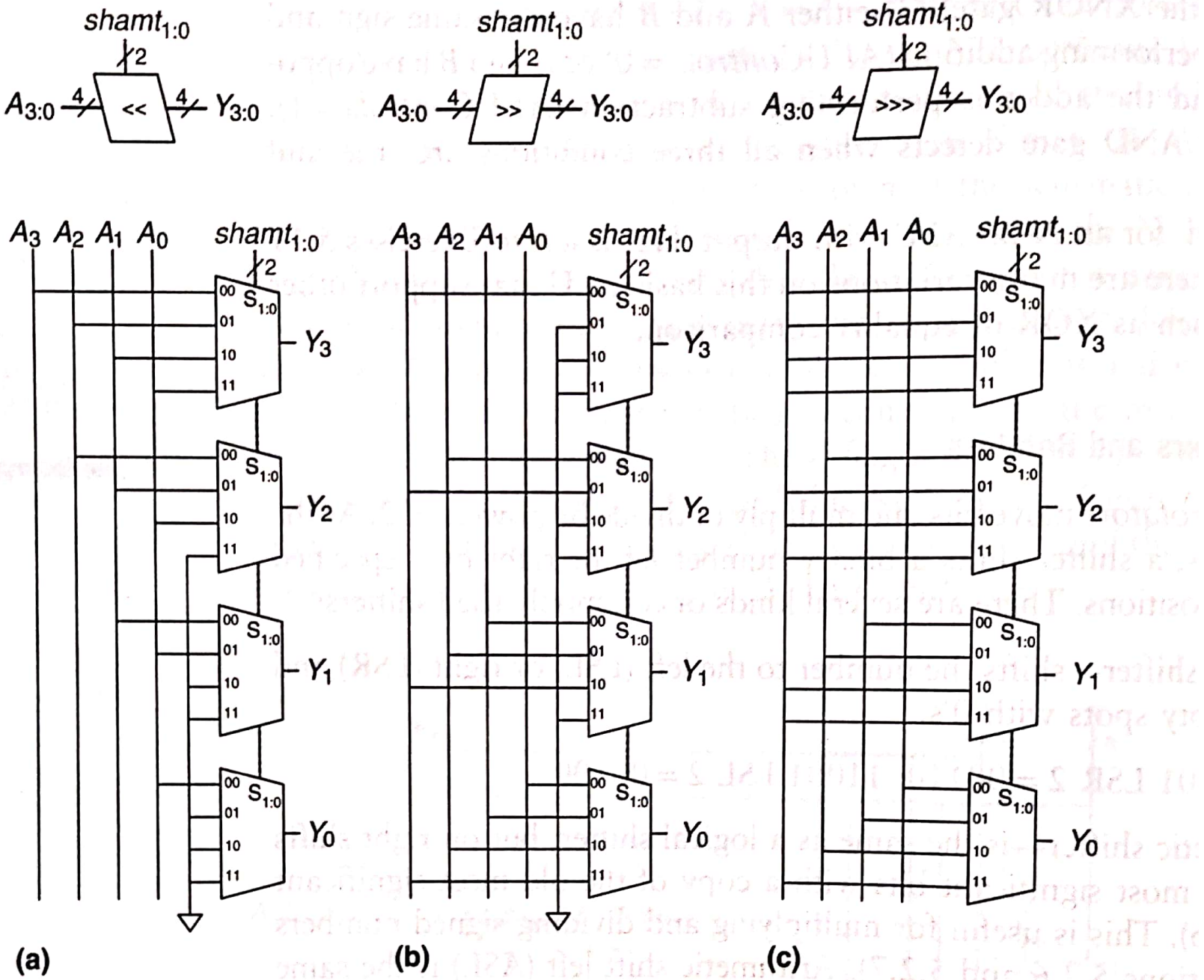


Figure 5.18 4-bit shifters: (a) shift left, (b) logical shift right, (c) arithmetic shift right

The behavior of the ALU Decoder is given by the truth tables in Table 7.3. For data-processing instructions, the ALU Decoder chooses *ALUControl* based on the type of instruction (ADD, SUB, AND, ORR). Moreover, it asserts *FlagW* to update the status flags when the *S*-bit is set. Note that ADD and SUB update all flags, whereas AND and ORR only update the *N* and *Z* flags, so two bits of *FlagW* are needed: $FlagW_1$ for updating *N* and *Z* ($Flags_{3:2}$), and $FlagW_0$ for updating C and *V* ($Flags_{1:0}$). $FlagW_{1:0}$ is killed by the Conditional Logic when the condition is not satisfied ($CondEx = 0$).

Table 7.2 Main Decoder truth table

Op	$Funct_5$	$Funct_0$	Type	Branch	MemtoReg	MemW	ALUSrc	ImmSrc	RegW	RegSrc	ALUOp
00	0	X	DP Reg	0	0	0	0	XX	1	00	1
00	1	X	DP Imm	0	0	0	1	00	1	X0	1
01	X	0	STR	0	X	1	1	01	0	10	0
01	X	1	LDR	0	1	0	1	01	1	X0	0
10	X	X	B	1	0	0	1	10	0	X1	0

Table 7.3 ALU Decoder truth table

ALUOp	$Funct_{4:1}$ (*cmd*)	$Funct_0$ (*S*)	Type	$ALUControl_{1:0}$	$FlagW_{1:0}$
0	X	X	Not DP	00 (Add)	00
1	0100	0	ADD	00 (Add)	00
		1			11
	0010	0	SUB	01 (Sub)	00
		1			11
	0000	0	AND	10 (And)	00
		1			10
	1100	0	ORR	11 (Or)	00
		1			10