Sample Instructions

LDR **r0,**address — ***Load*** the contents of the memory location at address into register **r0**.

STR r0,**address** — ***Store*** the contents of register r0 at the specified **address** in memory.

ADD **r0**,r1,r2 — ***Add*** the contents of register r1 to the contents of register r2 and store the result in register **r0**.

SUB **r0**,r1,r2 — ***Subtract*** the contents of register r2 from the contents of register r1 and store the result in register **r0**.

BPL target — ***If*** the result of the previous operation was ***plus*** (***+ve*** or ***zero***) ***then*** branch to the instruction at address target.

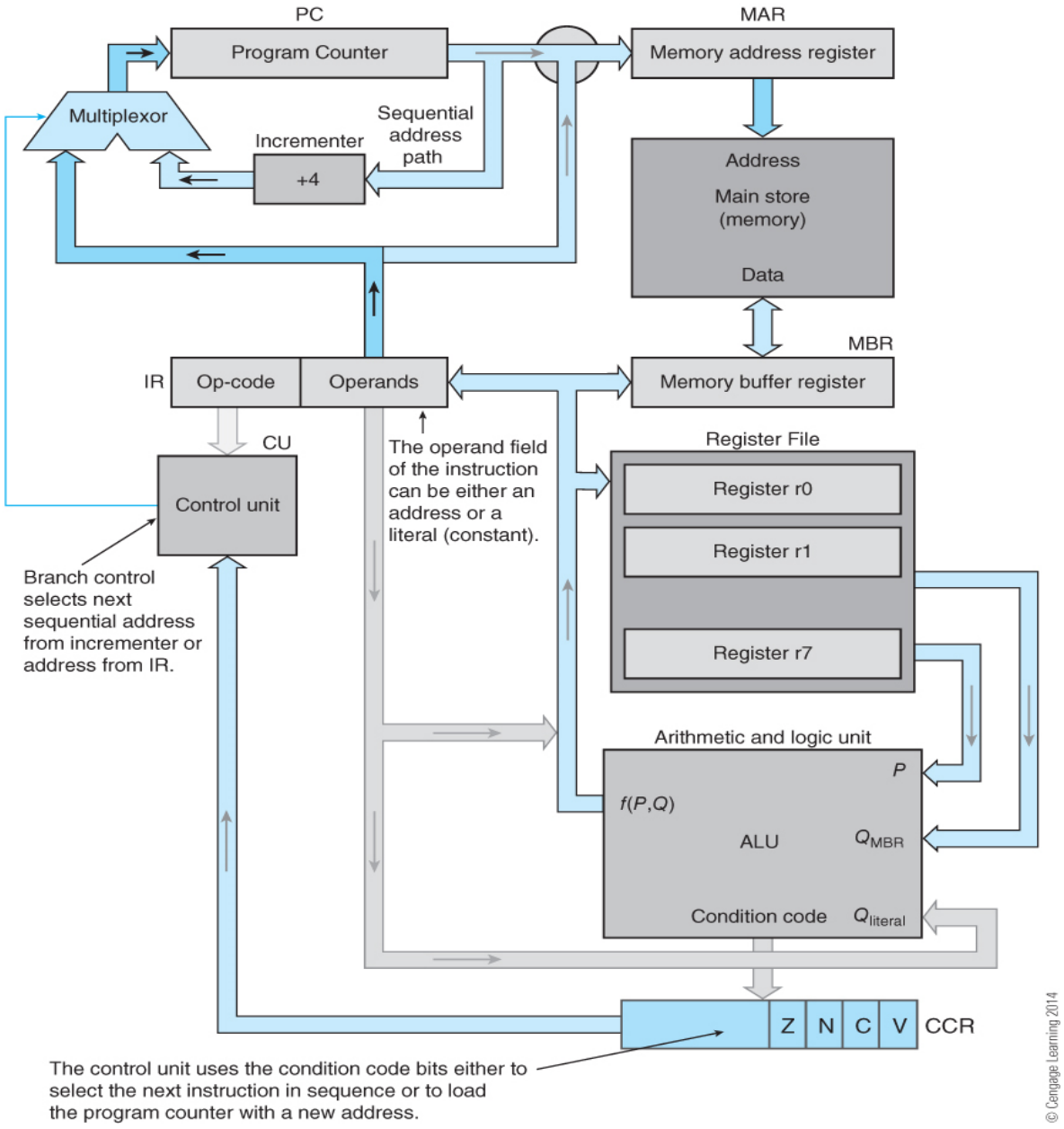
BEQ target — ***If*** the result of the previous operation was ***zero***, ***then*** branch to the instruction at address target.

B target — ***Branch unconditionally*** to the instruction stored at the memory address target.

Note the number of operands in each instruction.

Flow Control

FIGURE 3.5 Implementing conditional behavior at the machine level

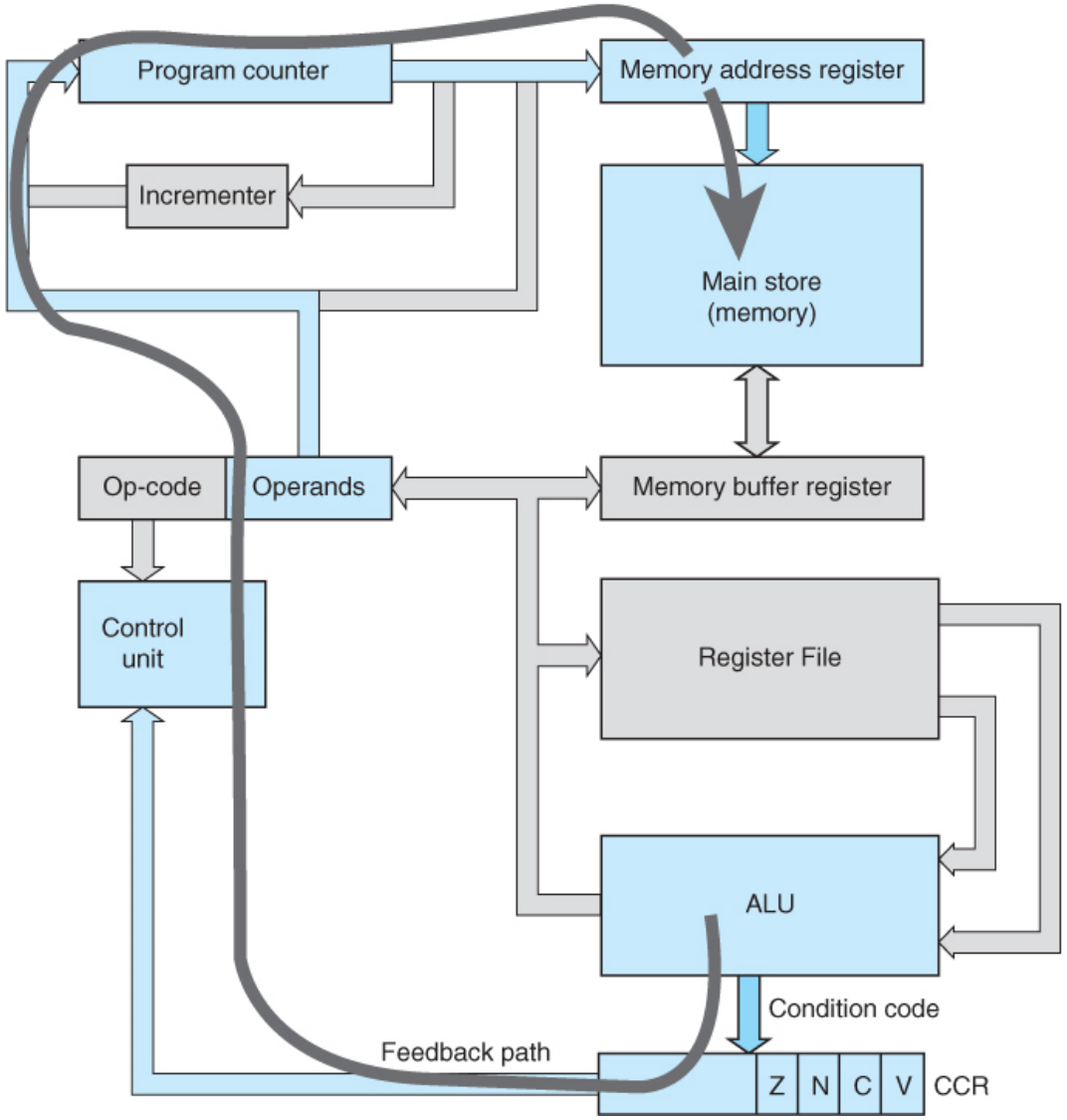


- *Flow control* refers to any action that *modifies* the strict instruction-by-instruction sequence of a program.
- *Conditional behavior* allows a processor to select one of two possible courses of action.
- A *conditional instruction* like BEQ results in either
  - continuing program execution normally, or
  - loading the program counter with a new value and executing a *branch* to another region of code.

Flow Control

Figure 3.6 illustrate how the result from the ALU can be used to modify the sequence of instructions.

FIGURE 3.6 Feedback from ALU to instruction



Status Bits (Flags)

- When the computer performs an operation, it stores *status* or *condition* information in the *Condition Code Register* (***CCR***).
- The processor records whether the result is
  - **Z**ero (Z),
  - **N**egative in two's complement terms (N),
  - generated a **C**arry (C), or
  - arithmetic o**V**erflow (V).

Status Bits (Flags)

- Example:

```
 00110011      11111111      01011100      11011100
+01000010     +00000001     +01000001     +11000001
 01110101      100000000     10011101      110011101
Z = 0, N = 0  Z = 1, N = 0  Z = 0, N = 1  Z = 0, N = 1
C = 0, V = 0  C = 1, V = 0  C = 0, V = 1  C = 1, V = 0

  51            -1            92           -36
 +66           +1            +65          -63
 ---           ---           ---          ---
 117            0            -99          -99
```

CISC means COMPLEX Instruction Set Computer

- CISC processors, like the *Intel IA32*
  - update status flags after each operation.

RISC means REDUCED Instruction Set Computer

- RISC processors, like the *ARM*,
  - require the programmer to request updating the status flags.
- The *ARM* does it *by appending an **S** to the instruction*;
  - for example SUBS or ADDS.

Example of a Conditional Operation

```
          SUBS  r5,r5,#1   ;Subtract 1 from r5
          BEQ   onZero     ;IF zero then go to the line labeled 'onZero'
notZero   ADD   r1,r2,r3   ;ELSE continue from here
      .
      .
onZero    SUB   r1,r2,r3   ;Here's where we end up if we take the branch
```

Explanation

SUBS r5,r5,#1

- subtracts 1 from the contents of register r5.
- After completing this operation the number remaining in r5 may, or may not, be zero.

BEQ onZero

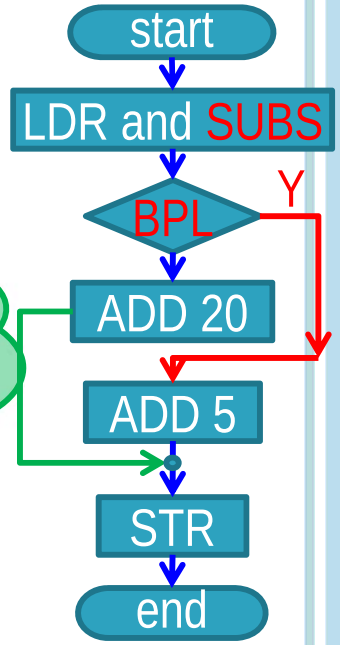
- forces a branch to the line labeled '**onZero**' if the outcome of the last operation was zero.
- Otherwise the next instruction in sequence after the BEQ is executed.

Example of a Conditional Operation

P ≥ Q is the same as P - Q ≥ 0

IF P ≥ Q THEN X = P + 5
ELSE X = P + 20

This is an unconditional branching that prevents the following instruction from being executed.

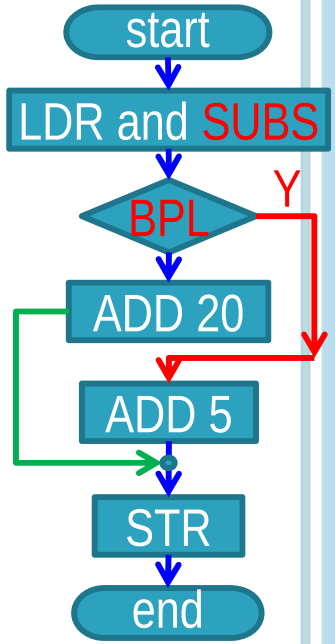



```
        LDR   r0,P      ;Load r0 with the contents of location P
        LDR   r1,Q      ;Load r1 with the contents of location Q
        SUBS  r2,r0,r1  ;Subtract the contents of Q from P
        BPL   THEN      ;IF P – Q ≥ 0 then execute the 'THEN' part
        ADD   r0,r0,#20 ;ELSE Add 20 to the contents of r0 to get P + 20
        B     EXIT      ;Skip past 'THEN' part to 'EXIT'
THEN    ADD   r0,r0,#5  ;Add 5 to r0 to get P + 5
EXIT    STR   r0,X      ;Store r0 in memory location X
        STOP
P       DCD   12        ;These three lines reserve memory space for
Q       DCD   9         ;the three operands P, Q, X. The memory
X       DCD   0         ;locations are 36, 40, and 44, respectively.
```

Here's where the test and conditional branching take place

DCD means, Define Constant Data

Example of a Conditional Operation



IF P ≥ Q THEN X = P + 5
ELSE X = P + 20

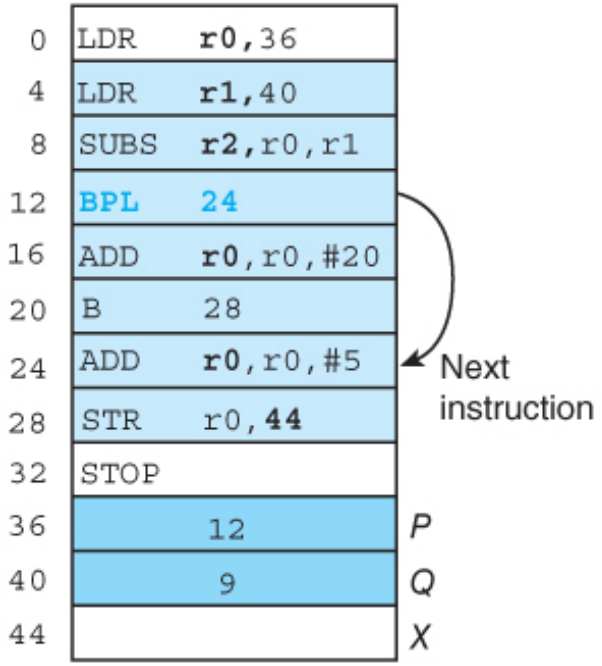
Same example, but with RTL comments

```
        LDR   r0,P       ;[r0]  ← [P]
        LDR   r1,Q       ;[r1]  ← [Q]
        SUBS  r2,r0,r1   ;[r2]  ← [r0] - [r1]
        BPL   THEN       ;IF [r2] ≥ 0 [PC] ← THEN
        ADD   r0,r0,#20  ;[r0]  ← [r0] + 20
        B     EXIT       ;[PC] ← EXIT
THEN    ADD   r0,r0,#5   ;[r0]  ← [r0] + 5
EXIT    STR   r0,X       ;[X]   ← [r0]
        STOP
P       DCD   12         ;These three lines reserve memory space for
Q       DCD   9          ;the three operands P, Q, X. The memory
X       DCD   0          ;locations are 36, 40, and 44, respectively.
```

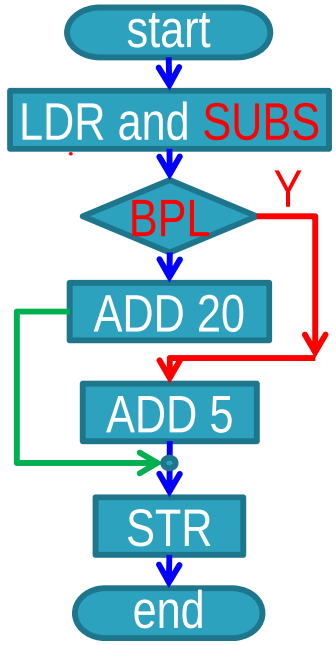
How are the locations P, Q, and X calculated?

Example of a Conditional Operation

Case 1: P = 12, Q = 9, and hence the conditional branch is *taken* *(i.e., will branch to THEN)*




```
                LDR   r0,P      ;[r0] ← [P]
                LDR   r1,Q      ;[r1] ← [Q]
                SUBS  r2,r0,r1  ;[r2] ← [r0] - [r1]
                BPL   THEN      ;IF [r2] ≥ 0 [PC] ← THEN
                ADD   r0,r0,#20 ;[r0] ← [r0] + 20
                B     EXIT      ;[PC] ← EXIT
THEN            ADD   r0,r0,#5  ;[r0] ← [r0] + 5
EXIT            STR   r0,X      ;[X] ← [r0]
                STOP
P               DCD   12
Q               DCD   9
X               DCD   0
```



PC_{start} = 0, PC_{end} = 4, r0 = 12
PC_{start} = 4, PC_{end} = 8, r1 = 9
PC_{start} = 8, PC_{end} = 12, r2 = 3
PC_{start} = 12, PC_{end} = 24,
PC_{start} = 24, PC_{end} = 28, r0 = 17
PC_{start} = 28, PC_{end} = 32, X = 17

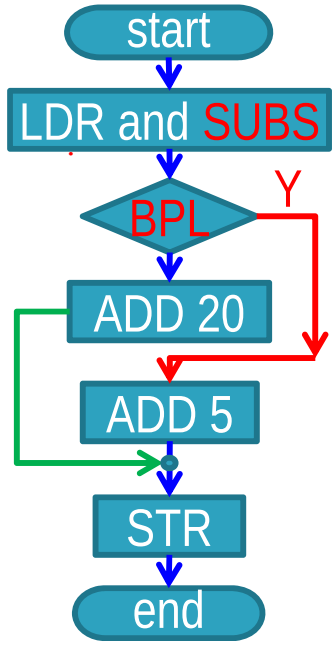
Example of a Conditional Operation

Case 2: P = 12, Q = 14, and hence the conditional branch is *not taken* *(i.e., will NOT branch to THEN)*

0	LDR	**r0**,36		
4	LDR	**r1**,40		
8	SUBS	**r2**,r0,r1		
12	BPL	24		
16	ADD	**r0**,r0,#20		
20	B	28		
24	ADD	**r0**,r0,#5		
28	STR	r0,**44**		
32	STOP			
36		12	*P*	
40		14	*Q*	
44			*X*	

Next instruction

```
                LDR   r0,P      ;[r0] ← [P]
                LDR   r1,Q      ;[r1] ← [Q]
                SUBS  r2,r0,r1  ;[r2] ← [r0] - [r1]
                BPL   THEN      ;IF [r2] ≥ 0 [PC] ← THEN
                ADD   r0,r0,#20 ;[r0] ← [r0] + 20
                B     EXIT      ;[PC] ← EXIT
THEN            ADD   r0,r0,#5  ;[r0] ← [r0] + 5
EXIT            STR   r0,X      ;[X] ← [r0]
                STOP
P               DCD   12
Q               DCD   14
X               DCD   0
```



PC_{start} = 0, PC_{end} = 4, r0 = 12
PC_{start} = 4, PC_{end} = 8, r1 = 14
PC_{start} = 8, PC_{end} = 12, r2 = -2
PC_{start} = 12, PC_{end} = 16,
PC_{start} = 16, PC_{end} = 20, r0 = 32
PC_{start} = 20, PC_{end} = 28,
PC_{start} = 28, PC_{end} = 32, *X* = 32

Example of a Conditional Operation

- Consider the code needed to calculate 1 + 2 + 3 + 4 + … + 20

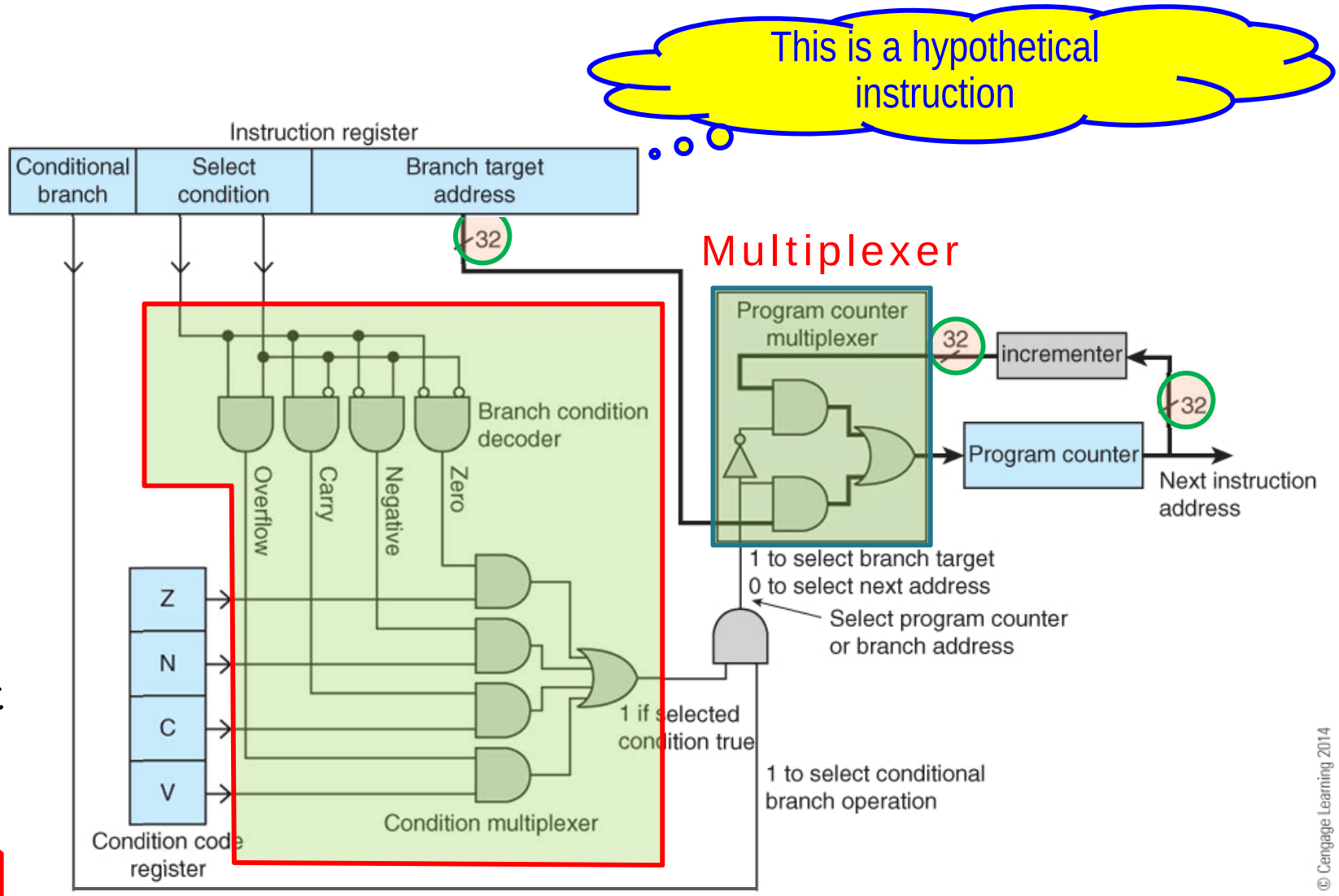
```
        MOV  r0,#1      ;Put 1 in register r0 (the counter)
        MOV  r1,#0      ;Put 0 in register r1 (the sum)
Next    ADD  r1,r1,r0   ;REPEAT: Add current counter to sum
        ADD  r0,r0,#1   ;    Add 1 to the counter
        CMP  r0,#21     ;    Have we added all 20 numbers?
        BNE  Next       ;UNTIL we have made 20 iterations
        STOP            ;If we have then stop
```

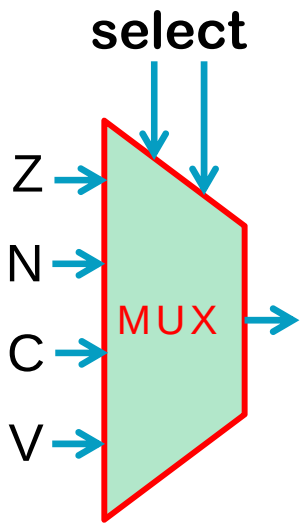
MOV and CMP instructions need ONLY two operands.

CMP compares the value in a register with *Operand2*, i.e., subtracting *Operand2 from the register value.*

It *updates* the condition flags on the result, but does *not* place the result in any register.

Example of a Conditional Operation





Multiplexer

Memory-to-memory vs Register-to-register instructions

- Figure 3.8a illustrates an instruction that implements ADD A,B,C where A, B, and C are 32-bit memory addresses.
  The width is 112 bits which is unfeasibly large.
- Figure 3.8b illustrates the format of a hypothetical RISC processor with a *register-to-register* format that can execute ADD R1,R2,R3 where the registers are chosen from 32 possible registers (requiring a 5-bit register address field).
  - Such a format is used by most 32-bit RISC processors with small variations.

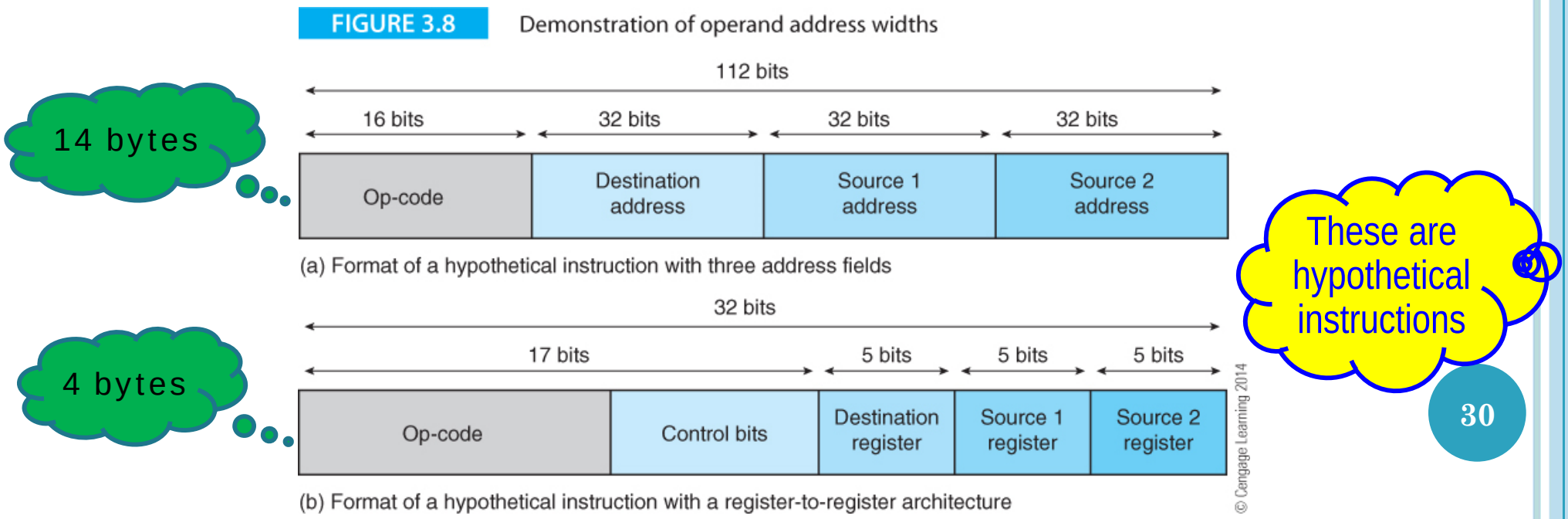


FIGURE 3.8 Demonstration of operand address widths

These are hypothetical instructions

General-Purpose Registers

- Registers usually have the same width as the fundamental word of a computer (*but not always so*).

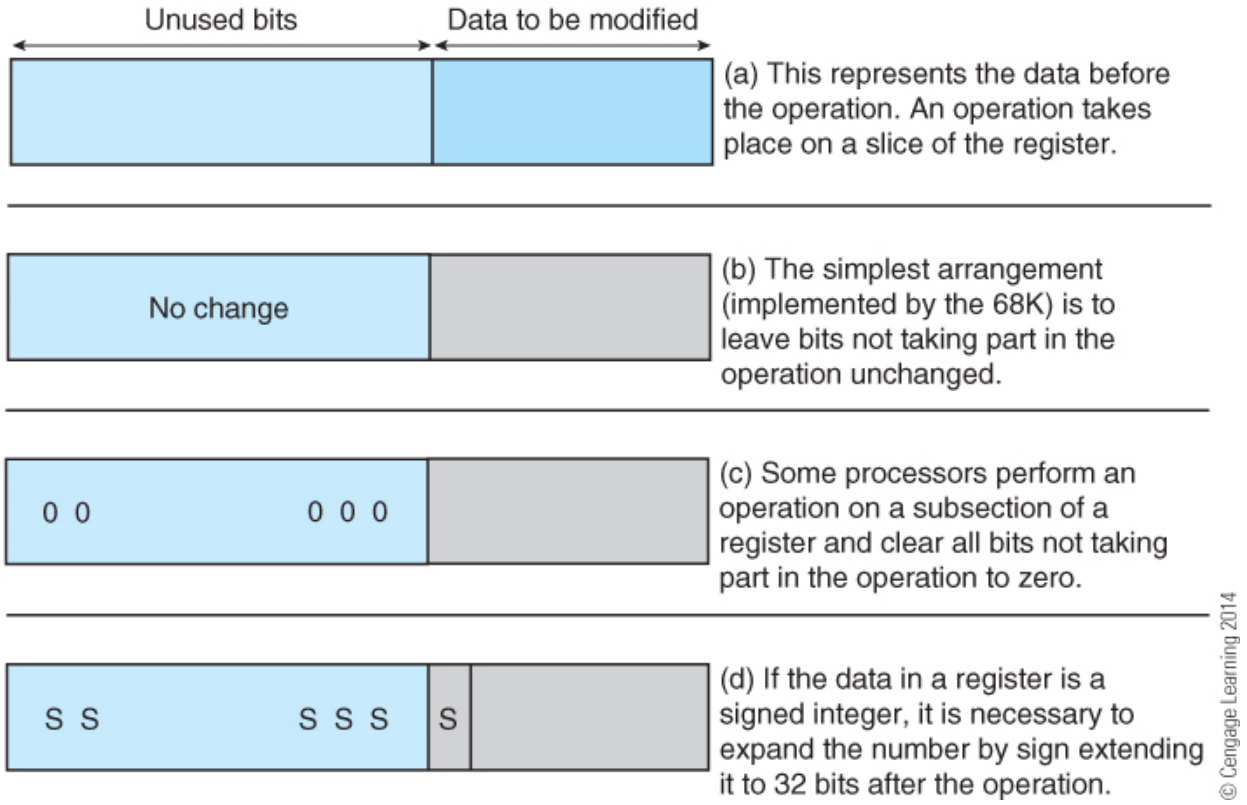
- Computers might have
  - *special-purpose* (dedicated) registers
  - *general-purpose registers*

- The ***ARM*** processors have
  - general-purpose registers, and
  - two special purpose registers (have special hardware-defined functions)
    - cannot be used by the programmer for general-purpose data processing.

Data Extension

- Sometimes registers hold data values smaller than their actual length
  - for example a 16-bit (halfword in a 32-bit word register).
- What happens to the other bits? (*processor dependent*)
  - some leave the unused bits unchanged,
  - some set the unused bits to 0, and
  - some sign-extend the 16-bit halfword to 32-bits (*two's complement*)

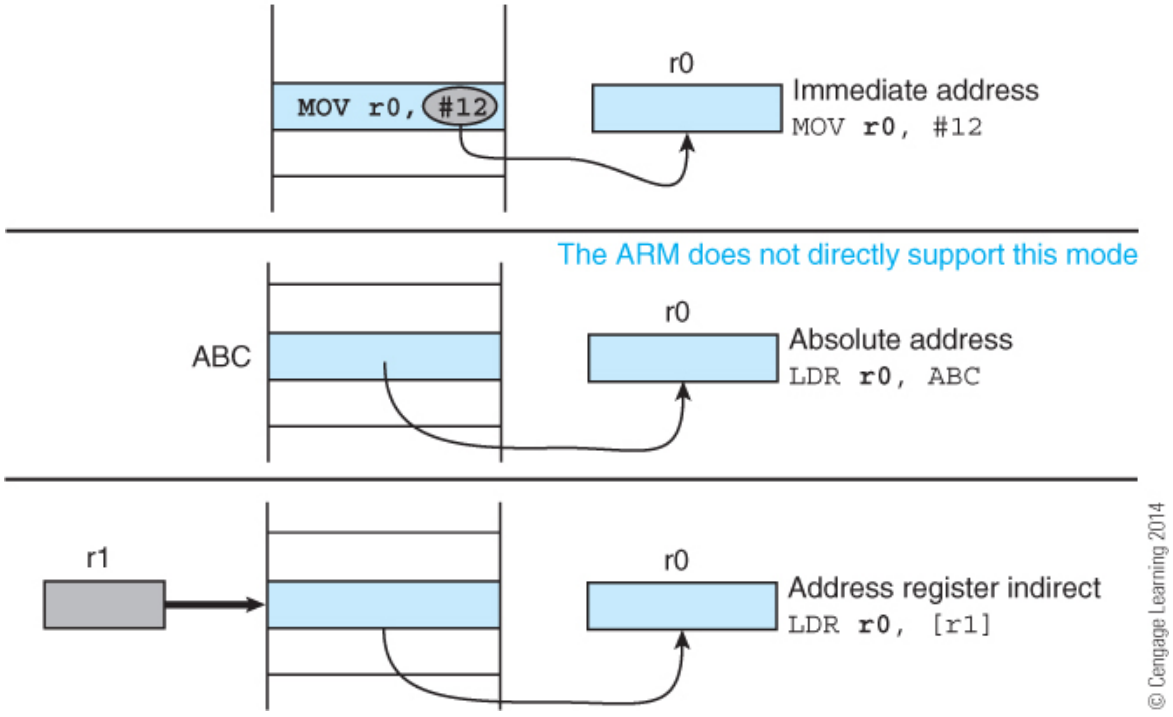
FIGURE 3.9 Operations on a subsection of a register



Addressing Modes

- There are three fundamental addressing modes
  - ***Literal or immediate***
    - the actual value is part of the instruction
  - ***Direct or absolute***
    - the instruction provides the memory address of the operand
    - *The ARM architecture does not directly support this mode*
  - ***Register indirect or pointer based or indexed***
    - a register contains the address of the operand

FIGURE 3.10 Progressive sequence of addressing modes



Instruction Types

- ***Memory-to-register***
  - The source operands are in memory and
  - the destination operand is in a register
- ***Register-to-memory***
  - The source operands are in registers and
  - the destination operand is in memory
- ***Register-to-register***
  - Both operands are in registers.

CISC means COMPLEX Instruction Set Computer

- ***CISC processors*** like the Intel IA32 family and Motorola/Freescale 68K family *allow memory-to-register* and *register-to memory data-processing* operations.

RISC means REDUCED Instruction Set Computer

- ***RISC processors*** like the ARM and MIPS *allow* ***only*** *register-to-register data-processing* operations.
- ***RISC processors*** *have* a *special LOAD* and a *special STORE* instructions (***pseudo instructions***) *to transfer data between memory and a register*.

Operands and Instructions

- ***Processors*** can have
  - *three-address* instructions,
  - *two-address* instructions,
  - *one-address* instructions, and
  - *zero-address* instructions.

- ***CISC*** processors *typically* have
  - ***Two-address*** instructions where
    - ***one*** address is ***memory*** and the ***other*** is a ***register***.

- ***RISC*** processors *typically* have a
  - ***three-address*** data processing instruction where
    - ***the three*** operand addresses ***are registers***.

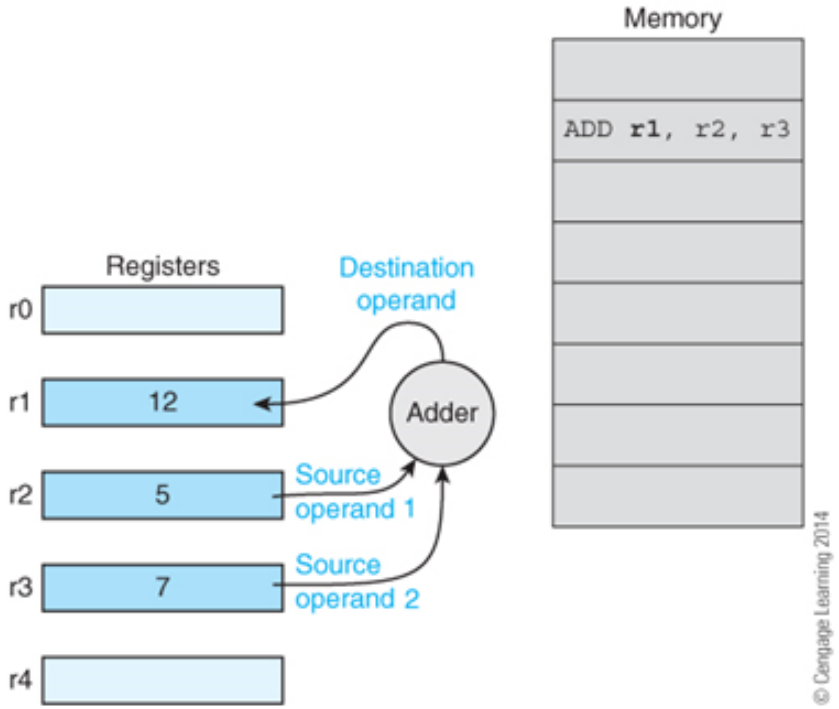
  - They *also have* ***two*** special *two-address* instructions,
    - **LOAD** and **STORE**.

Three Address Machines

- Processors *do not* implement *three memory address* instructions (***why?***)
- A typical RISC processor allows *three register addresses* in an instruction
  - For example:

ADD **r1**,r2,r3 ;Add r2 to r3 and put the result in **r1**

FIGURE 3.11 The three address instruction



Two Address Machines

- Typically, the *operands* are either
  - two registers or
  - one a register and one a memory location;

For example, the **ADD** instruction in a ***68K processor*** can be written as:

Instruction	RTL definition	Mode
ADD D0,**D1**	;[D1] ← [D1] + [D0]	Register-to-register
ADD P,**D2**	;[D2] ← [D2] + [P]	Memory-to-register
ADD D7,**P**	;[P] ← [P] + [D7]	Register-to-memory

- The ***price*** of a *two-operand* instruction format is
  - the *destruction* (*overwriting)* of one of the source operands.
- A ***CISC*** has a *two-address* instruction format.
  - You can execute Q ← Q + P.
  - One operand appears *twice*,
    - first as a source and then
    - as a destination.

One Address Machines

- A *one-address* machine specifies just one operand in the instruction.
- The *second operand* is a fixed register called an *accumulator* that doesn't have to be specified.
- For example, the operation one-address instruction ADD P means [A] ← [A] + [P]. The notation [A] indicates the contents of the accumulator.
- The simple operation R = P + Q can be implemented by the following fragment of 8-bit code from a first-generation 6800 8-bit processor.

```
LDA   P    ;load accumulator with P
ADD   Q    ;add Q to accumulator
STA   R    ;store accumulator in R
```

Zero Address Machines

- A *zero-address* machine uses instructions that *do not have an address at all*.
- A *zero-address* machine operates on data that is at the top of a stack
  - normally referred to as *stack machines*.
- A *pure zero-address* machine is not practical, as load and store instructions are needed to read and put date from and to memory
- Example: the code used to evaluate the expression Z = (A + B)*(C – D) might be written as:

PUSH A	Push A on stack
PUSH B	Push B on stack
ADD	Add top two items and push A+B on the stack
PUSH C	Push C on the stack
PUSH D	Push D on the stack
SUB	Subtract top two items and push C – D on the stack
MUL	Multiply top two items on stack (C - D), (A + B) push result
POP Z	Pull the top item off the stack (the result)

Zero Address Machines

- Stack machines can easily handle Boolean logic.

- Example, consider if (A < B) or (C == D).
  This can be expressed as:

PUSH A	Push A on stack
PUSH B	Push B on stack
LT	Pull A and B and perform comparison. Push true or false
PUSH C	Push C
PUSH D	Push D
EQ	Push C and D and test for equality. Push true or false
OR	Pull top two Boolean values off stack. Perform OR push result.

- The Boolean value on the stack can be used with a branch on true or a branch on false command as in the case of any other computer.