Executing conditional instructions

Our design to this point has been the SISD (single instruction single data) architecture which executes programs in a purely sequential manner. This is fine if we don't need to make decisions based on comparisons (ie. if , loops etc.)

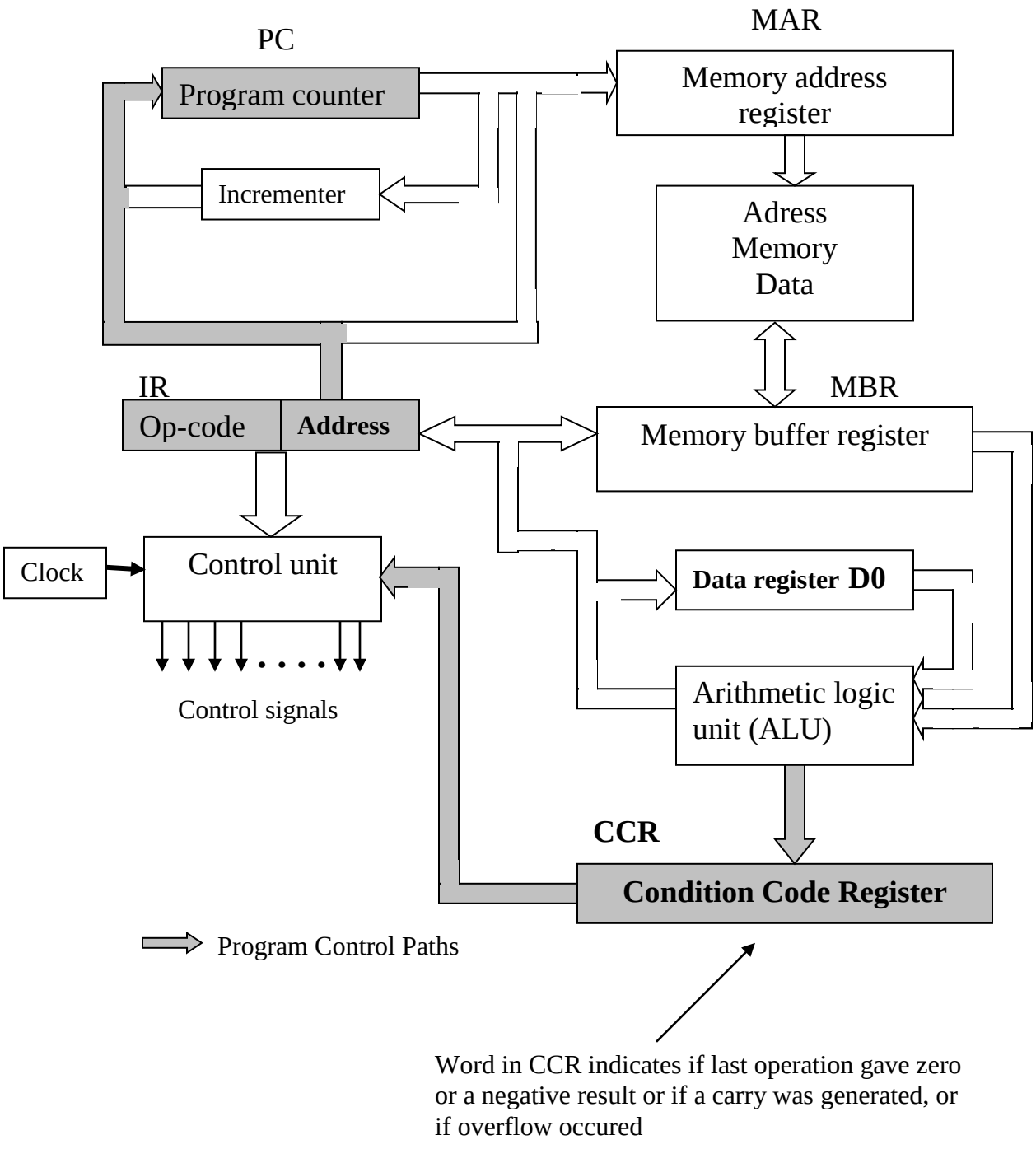
We now wish to describe a CPU that can execute conditional branches or jumps. This will allow our computer to process such familiar statements as the

if *conditional* then
    statement-block
else
    alternate statement-block

We add the following to our design

- A CCR – Conditional Code Register (or PSR – processor status register)
- A path between the CCR and Control Unit
- A path between the address field of the instruction register and the program counter

Our modified CPU looks like :


PC
MAR
Program counter
Memory address register
Incrementer
Adress Memory Data
IR
MBR
Op-code
Address
Memory buffer register
Clock
Control unit
Data register D0
Arithmetic logic unit (ALU)
Control signals
CCR
Condition Code Register
Program Control Paths
Word in CCR indicates if last operation gave zero or a negative result or if a carry was generated, or if overflow occured

The CCR looks to the ALU after each instruction is executed. When a conditional branch instruction is encountered the CCR's current state is looked at and the Control Unit forces either the next instruction in series or a branch to another instruction somewhere in the program.

Bits in the CCR are updated after each arithmetic or logic operation:

C= Carry	Set if a carry was generated by the last operation. This is the carry-out bit in the carry flip-flop (pg 174)
Z=Zero	Set if the last operation generated a zero result
N=Negative	Set if the last operation generated a negative result in 2's complement
V=O**v**erflow	Set if the last operation resulted in an arithmetic overflow (This can be detected, for instance, in addition if the sign bit of the result is different from the sign bit of both operators.)

The carry bit can be thought of as a 1 bit extension of the data register. When the ALU performs an arithmetic operation the result goes to the data register and the carry or borrow is retained in the carry-bit of the CCR. With shifts right or left by one bit, a bit *falls of the edge*, the bit is transferred to the carry-bit of the CCR.

Some examples of 8-bit addition and the Condition Code Register:

Operand 1		Operand 2		Result	CCR status bits
00000011	+	00000100	=	00000111	C=0, Z=0, N=0,V=0
11111111	+	00000001	=	00000000	C=1, Z=1, N=0,V=0
01100110	+	00110010	=	10011000	C=0, Z=0, N=1,V=1
11001001	+	10100000	=	01101001	C=1, Z=0, N=0,V=1

At times we perform operations which will update the CCR even though the update has no physical meaning –

Suppose we add the ASCII code for 'A' (65_{10}) which is %01000001 and the ASCII code for 'D' (68_{10}) which is %01000100 we get %10000101 and the negative bit gets set, N=1. Clearly this information is meaningless in context of the operation.

The control unit is connected to the Condition Code Register and the address field of the Instruction Register has a path back to the Program Counter, allowing modification of the contents of the Program Counter.

The conditional branch instructions can test a bit (or several bits) to see if they are set. If not the next instruction is obtained in the normal way. If the bit or bits are set the next instruction can be obtained using the address in Instruction Register. We call the address specified by the branch instruction the **target address**.

Some Typical Conditional Branch Instructions:

Mnemonic	Branch	Condition	
BCC	Branch on carry clear	C=0	
BCS	Branch on carry set	C=1	
BEQ	Branch on zero result	Z=1	
BNE	Branch on non-zero result	Z=0	
BMI	Branch on minus result	N=1	
BPL	Branch on positive result	N=0	
BVC	Branch on overflow clear	V=0	
BVS	Branch on overflow set	V=1	
BGE	Branch on greater than or equal to	$N\overline{V} + \overline{N}V = 1$	Used in 2's complement
BGT	Branch on greater than	$NV\overline{Z} + \overline{N}\overline{V}\overline{Z} = 1$	Used in 2's complement
BHI	Branch if higher than	$\overline{C}\overline{Z} = 1$	Used unsigned arithmetic
BLE	Branch if less than	$Z + \overline{N}V + N\overline{V} = 1$	Used in 2's complement
BLS	Branch if lower than or the same	$C + Z = 1$	Used unsigned arithmetic

Example:

BCC – Branch on Carry Clear : This does a jump to the target address if the carry bit in the Condition Code Register is 0

The corresponding Register Transfer Language (RTL):

```
BCC adrs: IF [C]=0 THEN [PC]←[IR(adrs)]
```

BEQ – Branch on equal (zero result): This does a jump to the target address if the Z bit in the CCR is 1.

RTL:

```
BEQ adrs: IF [Z]=1 THEN [PC] ←[IR(adrs)]
```

This might appear in code as

```
        SUB  x, D0        subtract the contents of x from D0
        BEQ Last          If result=0 branch to 'Last' otherwise continue
        . . .
        . . .
        . . .
Last                      target address of branch
```

We need one additional modification to our CPU to allow us to deal with literal operands. Indeed the modification produces a CPU capable of executing any computer program. We need an additional data path between the operand field of the IR , the data register and the ALU (next slide)

Until now the operands for instructions such as ADD have been addresses. At times we want to have an actual value as an operand rather than an address –

X= Y+ 8 as opposed to X=Y+Z

If we wanted to add 8 to D0 we could always store the value 8 at some memory location, say 105 and then

ADD 105, D0

An alternate solution is to

ADD #8, D0 RTL: [D0]←[D0]+8

where #8 stands for the actual value (literal).

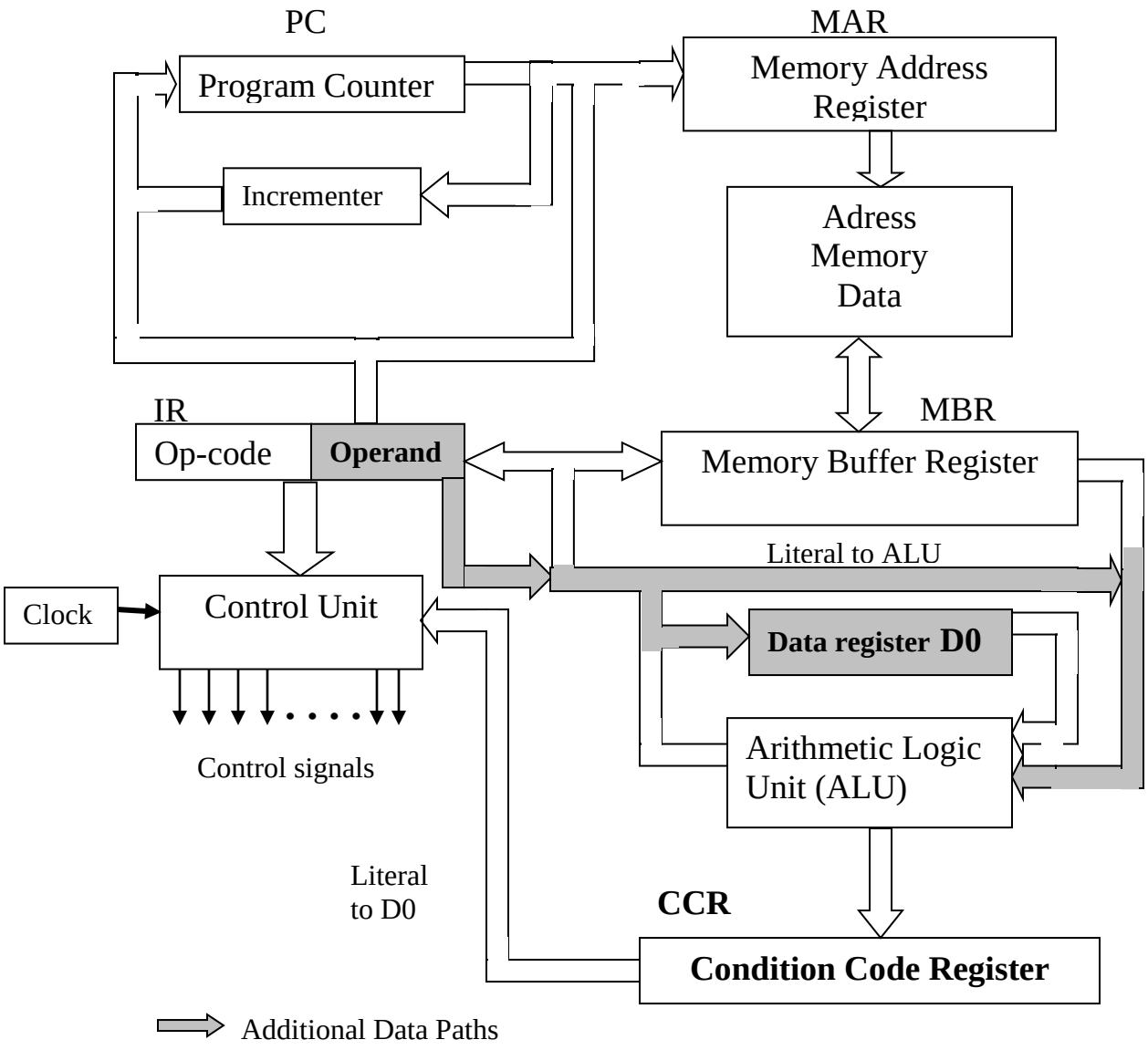
ADD #<literal> , <destination>

and

ADD <literal>, <destination>

are represented by different op-codes.

The added data path:



We now will describe a simple assembly language program on the computer we have been designing.

Suppose the one-address instruction set is as follows:

Op-code	Mnemonic	Action
00 M	MOVE M,D0	Load D0 with the contents of location M
01 M	Move D0,M	Store contents of D0 in M
02 M	IN M,D0	Input data from devive number M into D0
03 M	OUT D0,M	Output the contents of D0 to device number M
04 M	ADD M,D0	Add the contents of M to D0
05 M	SUB M,D0	Subtract the contents of M from D0
06 M	AND M,D0	Logical AND of contents of M with D0
07 M	OR M,D0	Logical OR of contents of M with D0
08	NEG D0	Complement contents of D0
09	ASL D0	Shift D0 one place left
0A	ASR D0	Shift D0 one place right
0B M	CMP M,D0	Compare contents of M with contents of D0
0C M	MOVE #M,D0	Put the number M into D0
0D N	BEQ N	Branch if Z-bit set to location N
0E N	BNE N	Branch if Z-bit clear to location N
OF N	BCC N	Branch if C-bit clear to location N
10 N	BCS N	Branch if C-bit set to location N
11 N	BRA N	Unconditional branch to location N
12	STOP	Stop

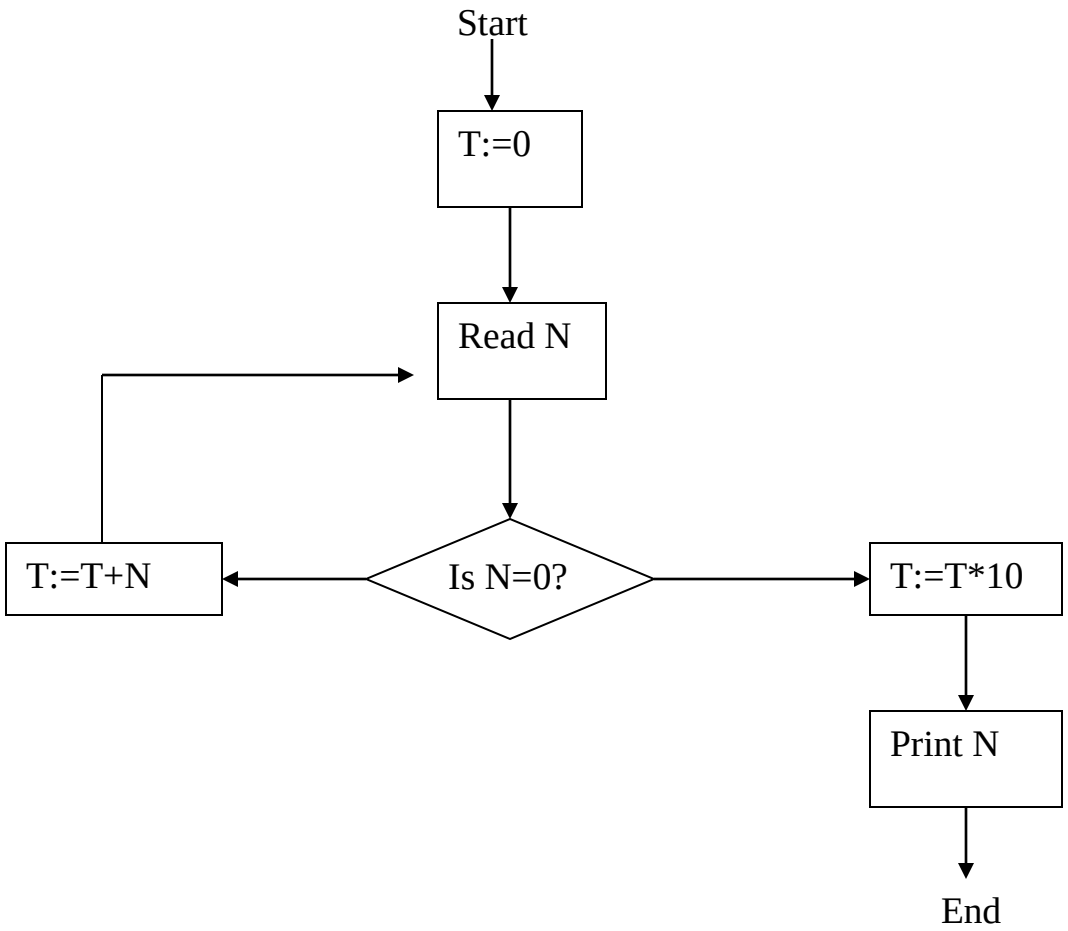
Note: We have not added IN and OUT data paths to our CPU yet.

We also will assume the keyboard is device number 0 and the monitor is device 1. We are also assuming these devices send and receive numbers, when reality they probably deal with ASCII-encoded data.

The program we wish to write is as follows:

Read a series of numbers from the keyboard that are terminated by a zero, and add them together, multiply the result by 10 and print the answer on the display.

A flow chart representing this might look like:



Flow charts used to be the norm for describing your programs before you started coding. They are excellent for describing simple algorithms, however, for complex projects they often confuse the structure one is describing. They also are not easily modifiable.

When starting an assembly language program it is a good idea to code the problem first in pseudocode or in some high level language such as C.

Note: It is the norm to describe algorithms (which are independent of language) in psuedocode form.

The C version of our program:

```
#include <stdio.h>
void main(void)
{
        int T=0, N;
        scanf("%i",&N);
        while (N !=0)
        {
                T=T+N;
                scanf("%i", &N);
        }
        T=T*10;
        printf("%i",T);
        return;
}
```

Pseudocode is an informal notation allowing the user to express algorithms in plain English (insert your favorite language here) using standard constructs found in structured programming.

A sample of someones pseudocode for the same program:

```
Module sum_times_ten
    Clear Total
    Repeat
        Get number
        Total:=Total +number
    Until number =0
    Total:=Total*10
    Print Total
End sum_times_ten
```

We encounter a minor problem when converting to assembly language, there is no command for direct multiplication. We recall from our knowledge of the representation of integers that if we shift our its left it is equivalent to multiplying by 2.

To calculate 10 T we observe

$$10\,T = 8T + 2T$$
$$= 2\times2\times2\times T + 2\times T$$
$$= 2\times(2\times2\times T + T)$$

Our Assembly Language Program:

Line #	Labels	Instructions	Comments
1.		NAM EXAMPLE	
2.		ORG $20	Data origin $20 (32 decimal)
3.	TOTAL	DS 1	Reserve a word of storage for TOTAL
4.		ORG 0	Origin of program
5.		MOVE #0, D0	[D0]←0
6.		MOVE D0,TOTAL	[TOTAL] ←[D0]
7.	REPEAT	IN 0,D0	[D0] ←Input
8.		BEQ MULT	If last result 0 branch to MULT
9.		ADD TOTAL,D0	[D0] ←[D0]+[TOTAL]
10.		MOVE D0,TOTAL	[TOTAL] ←[DO]
11.		BRA REPEAT	[PC] ←REPEAT
12.	MULT	MOVE TOTAL,D0	[D0] ←[TOTAL]
13.		ASL D0	[D0] ←[D0]*2
14.		ASL D0	[D0] ←[D0]*2
15.		ADD TOTAL,D0	[D0] ←[D0]+[TOTAL]
16.		ASL DO	[D0] ←[D0]*2
17.		OUT D0,1	Output ←[D0]
18.		STOP	
19.		END	

The first column is not part of the assembly language program, it is only for our reference.

The second column represents the label field. Labels are markers that may be referred to by other assembly language statements. When the assembler translates the code into machine language references to these labels are replaced by addresses of the lines they reference.

The third column holds assembler directives an assembly instruction code.

The fourth column is where our comments go. They will be ignored by the assembler.

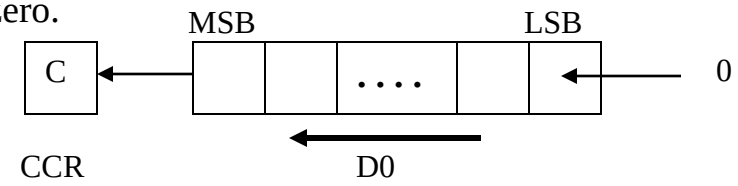
In the third column we see some mnemonics that are not part of our instruction set. NAM, ORG,DS these are the assembler directives.
They do not get translated into executable code.
NAM names the program
ORG sets the beginning or origin to $20 (decimal 32) It was a guesstimate that our code would fit in addresses 1-19 and data could be put from location $20 onward.
DS, define storage, reserves 1 word for TOTAL and binds it to the symbolic name. The C equivalent: int x ; ≡ x DS 1

END tells the assembler the end of the code has been reached.

The instruction

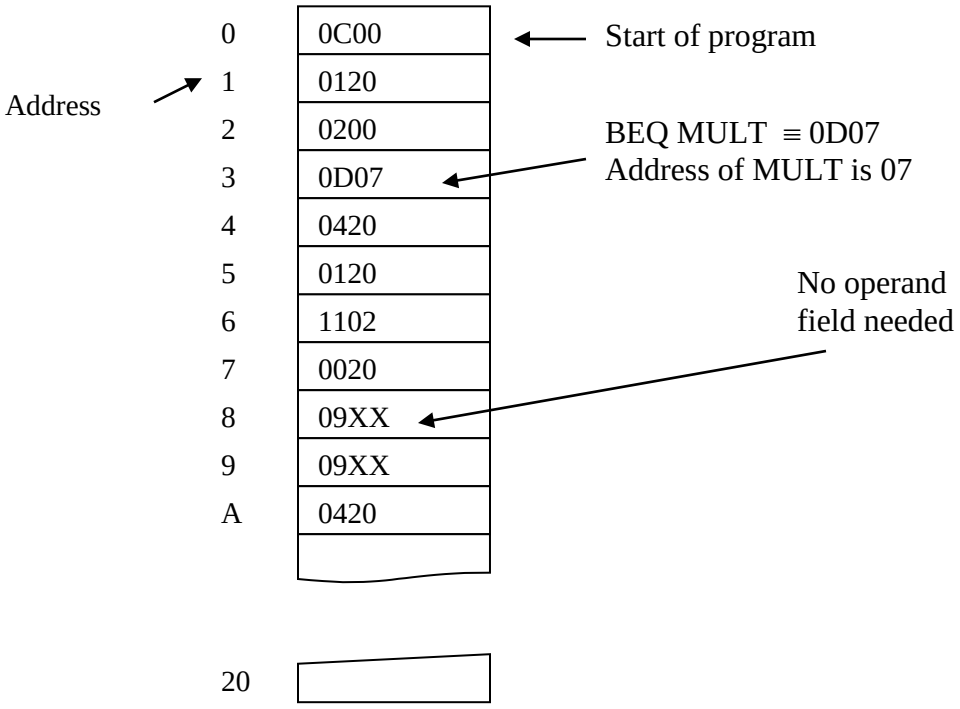
ASL D0

shifts the word in the registger D0 one unit left. The most significant bit gets copied to the C bit of the Condition Code Register. The least significant bit is occupied by zero.



Now that the program has been assembled and is in memory it is ready for execution.

A memory Map of our program *might* take the form:



Recall from our instruction set table

`0D N  BEQ N` Branch if Z-bit set to location N

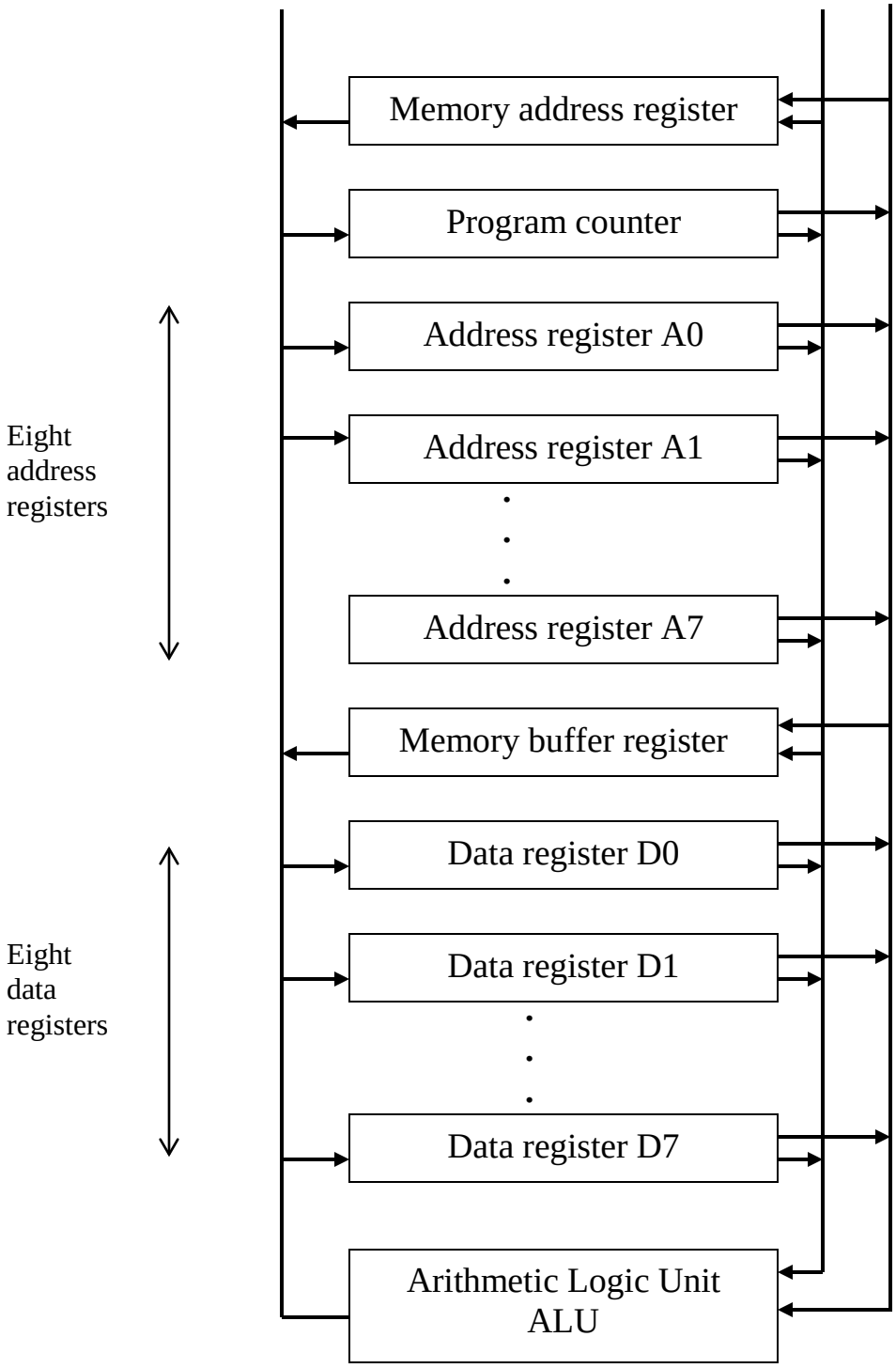
The architecture of a typical high-performance CPU

1970's - the 8 bit microprocessor with few internal registers.
1980's - the 16/32 bit microprocessor multiple internal registers.
1990's - Concern was with speed.

We wish to look at a model for a modern microprocessor. The model we look at will have multiple data registers, multiple address registers. It will still have the same basic registers as the more primitive CPU previously discussed.
It will also contain three internal buses to enable output from two registers to be fed to the ALU. The output from the ALU can be fed to any of the data or address registers. The set up will allow for operations of the form

$$R_3 = f(R_1,R_2)$$

where R1,R2 and R3 are any three registers and f ( ) is an operation performed by the ALU



Structure of CPU with multiple address and data registers

Now that we have added 16 user-accessible registers, we will require a more complex instruction format. We need a two-address instruction format requiring three fields:

op-code, operand address, and register address

Sampling of Mototola's 68000 microprocessor instructions

Mnemonic	Opreartion
ADD address,Di	[Di]←[Di] +[M(address)]
ASL #number,Di	Shift [Di] left by <number >bits
ASR #number,Di	Shift [Di] right by <number >bits
DIVU address,Di	[Di] ←[Di]/[M(address)]
MULU address,Di	[Di] ←[Di]*[M(address)]
NEG address	[M(address)] ←-[M(address)]
AND address,Di	[Di] ←[Di]·[M(address)]
OR address,Di	[Di] ←[Di]+[M(address)]
NOT address	$[M(address)] \leftarrow \overline{[M(address)]}$
SUB address,Di	[Di] ←[Di]-[M(address)]
TST address	[M(address)]- 0
CMP address,Di	[Di]-[M(address)]
EXG Ri,Rj	[Ri] ←[Rj], [Rj] ←[Ri]
MOVE address,Di	[Di] ←[M(address)]
MOVE Di,address	[M(address)] ←[Di]
MOVE #data,Di	[Di] ←data
MOVE #data,address	[M(address)] ←data
LEA address,Ai	[Ai] ←address
BRA address	[PC] ←address
BCC address	IF C=0 [PC] ←address

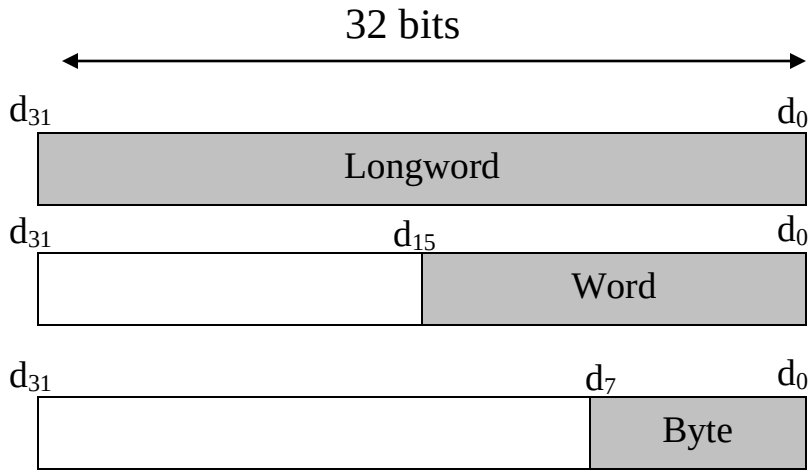
Data Registers of the 68000

8 data registers each 32 bits wide

You can operate on the whole register, 32 bits : a longword

or on the low-order 16 bits : a word

or on the low-order 8 bits: a byte



Structure of 68000's data registers

To indicate you wish to operate with bytes you append a .B to the mnemonic : `ADD.B D0,D1`

To indicate you wish to operate with words you append a .W to the mnemonic : `ADD.W D0,D1`

To indicate you wish to operate with longwords you append a .L to the mnemonic : `ADD.L D0,D1`

If you don't specify W is the default.

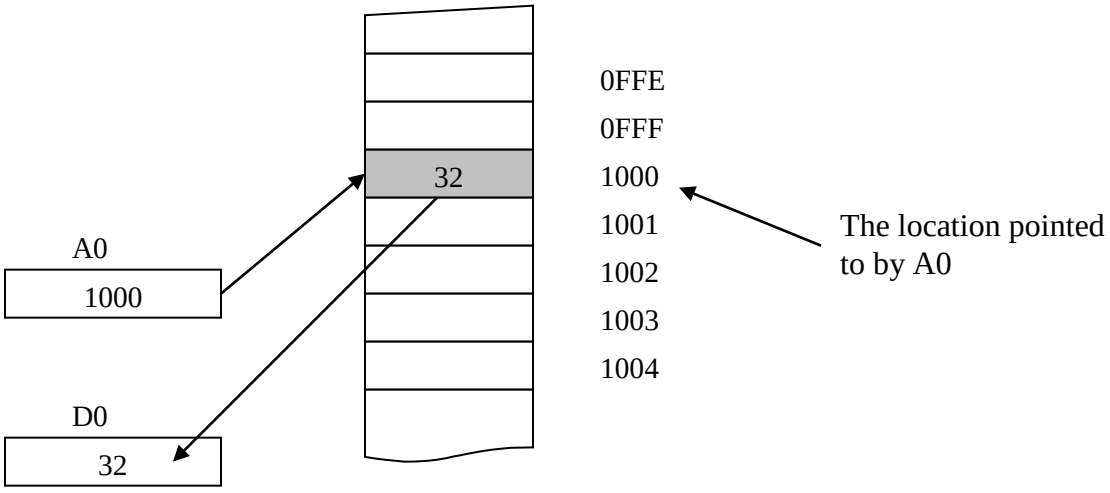
One of the advantages of having multiple data registers is that frequently used data can be stored on-chip making access extremely efficient (and fast).

Another major difference between the original simple CPU and this new version is the addition of the eight address registers. These registers contain the addresses of operands to be accessed in main memory. These registers are convenient for working with data structures such as arrays, lists, tales and vectors.

Typical usage:

```
MOVE.B (A0),D0          RTL: [D0]←[D0]+[M([A0])]
```

How we view it:



Notice we use parentheses to indicate this is the address of the operand. This addressing mode is called *address register indirect addressing*.