PROGRAMAR EM ASSEMBLY (PARTE I)

“Muitos utilizadores e a totalidade dos programadores deparará,tarde ou cedo, com a necessidade de resolver um problema específico de codificação para o qual as linguagens de alto nível não oferecem qualquer solução ou, se oferecem, esta não é satisfatória.”

PORQUÊ PROGRAMAR EM *ASSEMBLY* ?

Muitos utilizadores de computadores pessoais conhecem linguagens de programação como o BASIC, o PASCAL ou o C e dão-se por muito satisfeitos. Na realidade, para muitos objectivos de programação, ferramentas como essas podem ser, e são normalmente, mais do que suficientes, ou dito de outro modo, são efectivamente as mais adequadas.

Essas linguagens de programação são designadas por linguagens de «**alto nível**», o que significa que quem as utiliza programa a um elevado nível de distanciamento do equipamento. Outra forma de as designar é por linguagens «**procedurais**» pois permitem-nos descrever as tarefas a serem realizadas de forma orientada directamente para a resolução do problema.

Para elas não há qualquer preocupação com instruções aceites a nível do processador, registos internos e tudo que tenha a ver com a arquitectura particular de um computador.

Na verdade é possível programar em linguagem de alto nível e ter um conhecimento não muito pormenorizado do computador com que se está a trabalhar.

As linguagens de alto nível têm ainda a vantagem de ser relativamente «portáveis» entre computadores de arquitecturas diferentes, se é que se pode falar de portabilidade satisfatória nos nossos dias em que os programas tendem a ser grandes e complexos procurando tirar o maior partido dos computadores em que irão ser executados. Mas, como o leitor já adivinhou, nem tudo são rosas com as linguagens de alto nível. Se assim fosse, não só este artigo não faria sentido, como não se venderiam mundialmente milhares de obras sobre a outra linguagem, a linguagem de **baixo nível.**

Muitos utilizadores e a totalidade dos programadores deparará,tarde ou cedo, com a necessidade de resolver um problema específico de codificação para o qual as linguagens de alto nível não oferecem qualquer solução ou, se oferecem, esta não é satisfatória. A frequência de aparecimento desses problemas não é tão invulgar como à primeira vista possa parecer. Para um programador de aplicações de gestão podem aparecer ocasionalmente mas, para um programador de sistemas operativos, aparecerão a todo o momento. E a sua resolução passará normalmente por possuir um elevado conhecimento dos «meandros» de como o equipamento funciona e de como actuar sobre ele. A melhor via para realizar essa aprendizagem é começar a estudar a única linguagem que o computador entende.

Ora bem, mas todos sabemos que os computadores internamente – os seus componentes físicos e circuitos electrónicos, todo esse conjunto que se designa por *hardware*, só conhecem uma linguagem que é constituída por zeros e uns, a **linguagem binária ou código máquina.**

Felizmente, não nos devemos alarmar, pois longe vão os dias em que se programava directamente em binário. Nos nossos dias já ninguém o faz!

Um programa binário é praticamente inintelígivel para nós humanos, mesmo que contenha apenas 2 ou 3 instruções. Mas nos primeiros tempos da era dos computadores, nos anos 40 e 50, houve quem o fizesse, naqueles grandes computadores de válvulas que já passaram à História.

Tentou-se em alternativa programar em hexadecimal, que é uma base de numeração em que se começa a contar no zero e se termina no «F» (que significa 15 em numeração decimal). Base esta que tem uma relação simples com a base binária, pois por cada grupo de 4 dígitos binários temos 1 em hexadecimal. Porém, a programação em hexadecimal também era extraordinariamente complicada e demorada.

Finalmente encontrou-se a solução, o *Assembly*.

O *Assembly* é uma linguagem de símbolos, designados **mnemónicas**.

Cada mnemónica tem a sua correspondência em um comando elementar inteligível pelo computador. Cada mnemónica é uma **instrução** *Assembly*.

Normalmente todos os comandos que podem ser dados em binário podem ser dados em *Assembly* com a mesma eficiência. Mas talvez seja melhor, neste momento, dar uma pequena espreitadela ao quadro seguinte, sem qualquer preocupação especial de compreender:

Endereço	Hexadecimal	Instrução em *Assembly*
0600:0100	B4	MOV AH,20
0600:0101	20	
0600:0102	B7	MOV BH,30
0600:0103	30	
0600:0104	00	ADD AH,BH
0600:0105	FC	
0600:0106	80	CMP AH,B0
0600:0107	FC	
0600:0108	B0	
0600:0109	75	NZ 104
0600:010A	F9	
0600:010B	B4	MOV AH,4C
0600:010C	4C	
0600:010D	CD	INT 21
0600:010E	21	

Certamente que, se o leitor nada sabia de *Assembly*, continou sem saber após a visulização do quadro, mas não fique preocupado por causa disso, pois «Roma e Pavia não se fizeram num dia» e, no seguimento deste artigo e

nos seguintes, tudo começará a fazer sentido. O quadro que acabou de ver apresenta-nos um extracto de um programa muito simples.

O formato do quadro é algo parecido com o que nos apresenta o programa DEBUG que acompanha o sistema operativa MS-DOS quando carregamos na tecla 'U'. Mas vamos explicar um pouco o quadro:

Os valores da coluna de titulo **Hexadecimal** são directamente intelígiveis, ou melhor, o seu equivalente na base binária é directamente inteligível por um computador da família IBM PC. As mnemónicas *Assembly* da coluna **Instrução em *Assembly*** tem o mesmo objectivo, mas são inteligíveis por nós humanos, e apenas por nós. O programa acima executa como se segue:

1) Mover (note o **MOV** na coluna de instruções em *Assembly*) para um **registo** designado por **AH** o valor 20 (hexadecimal). E aqui o leitor pensará, e com razão, que não só não sabe o que é um registo como **AH** não lhe diz nada. O assunto será desenvolvido a seu tempo mas, por hoje, fique com a ideia que a UCP (Unidade Central de Processamento) do computador é dividida em pequenas unidades de armazenagem designadas por registos. Os registos são de 16 bits (2 bytes) nos processadores 8086 e 80286. Todos têm um nome.

Um deles chama-se Accumulator e recebe a abreviatura **AX**.

O registo **AX** pode ser controlado por inteiro ou por metades. Se forem os primeiros 8 bits designa-se registo **AL** (*Accumulator Low*), se forem os últimos designa-se por **AH** (*Accumulator High*). Repare que não se utilizou o termo subregisto.

Outro registo designa-se por **BX** (*Base Register*) e o que se disse para o **AX** aplica-se a ele, isto é, pode ser considerado dividido em dois registos de 8 bits, o **BH** e o **BL.**

Mas continuando a nossa análise:

2) Mover também, mas agora o valor 30 (hexadecimal) para o registo **BH**.

3) Somar (Diz-se **ADD** em inglês) o conteudo do registo **AH** com o conteúdo do registo **BH** e deixar o resultado em **AH.**

Naturalmente que na primeira iteração **AH** ficará com o valor 50 (hexadecimal).

4) Agora comparamos (Note **CMP** de *Compare*) se o conteúdo do registo **AH** é igual ou não a **B0** (corresponde a saber se é igual a 176 em decimal).

A comparação é feita subtraindo **AH** de **B0** sem devolver o resultado, mas apenas sinalizando para um registo especial de nome ***flags*** o que se passou. O registo ***flags*** é também de 16 bits e cada um desses bits ora fica a **0** ora a **1** consoante o que se passou na instrução que acabou de ser executada. No nosso caso se houver igualdade de **AH** com **B0** o sétimo bit do registo ***flags*** e que é designado por ***zero flag*** assume o valor **1**.

5) Chegamos agora a um ponto de decisão. Se a comparação não der igual saltamos de novo para a instrução existente no endereço 104 (e que já vimos qual era) para ser executada de novo. Se a comparação der igual seguimos em frente e executamos as duas instruções seguintes que terminam o programa e nos trazem de novo para o *prompt* do DOS.

A instrução **JNZ** (*JUMP IF NOT ZERO*) actuou, por conseguinte,verificando no registo ***flags*** se o resultado da instrução anterior foi zero, isto é se o conteudo de **AH** é igual a **B0**.

Experimente transcrever as mnemónicas acima para o DEBUG teclando 'A' e escrevendo-as uma a uma. Não se preocupe com os valores constantes das colunas de endereços ou de símbolos hexadecimais, mas apenas com as mnemónicas. Execute o programa em «câmara lenta» teclando sucesivamente 'P' e verifique que os valores de alguns registos se alteram após cada instrução.

Pode gravar o programa para uso posterior ou para executá-lo do *prompt* do DOS, e fazer muitas outras coisas com o DEBUG, pelo que vale bem a pena ler o manual do MS-DOS para ver como trabalha esta pequena pérola.

É possivel que, no decorrer das suas experiências com o DEBUG, o computador ocasionalmente «pendure» e tenha de carregar na tecla de *Reset*.

Normalmente isso é inconsequente, pois sei de pessoas que destruiram os dados do disco com outros programas que não o DEBUG (não conheço nenhuma mas é natural que existam).

Tenha contudo em atenção que alguns comandos do DEBUG são extremamente perigosos e não deverão ser utilizados sem uma perfeita compreensão da sua acção e do que se pretende atingir. Estou a lembrar-me de um, o que permite a escrita em sectores absolutas do disco – se possível nunca o use.

A aprendizagem das instruções de *Assembly* é fácil pois as mnemónicas estão organizadas por classes de instruções de código máquina de um modo que faz sentido para nós humanos. Por exemplo, todas as acções que implicam um movimento de bytes de um local de armazenamento para outro representam-se pela mnemónica **MOV** seguida de qualquer outra coisa.

O **MOV** pode ser de transferência entre registos do processador, de transferência da memória para os registos ou vice-versa, de carregamento de um valor constante para um registo, etc. Deste modo, com o conhecimento de umas poucas dezenas de mnemónicas, podemos realizar quase qualquer programa em *Assembly*.

E fique desde já consciente que a mnemómica mais usada é precisamente o **MOV** e que alguns programas chegam a ter mais de 50% de **MOV**'s.

Outra mnemónica muito usada é **ADD** e que já vimos também.

A mnemónica **CMP** é fundamental para a instrução que se lhe segue e que é normalmente uma instrução de «salto condicional para».

As instruções de salto condicional actuam verificando o que se passa nas ***flags*** . No nosso exemplo, e como se viu, foi o bit ***zero flag*** do registo ***flags*** que foi testado.

Uma ideia muito difundida, mas largamente errada, é a de que a programação em *Assembly* é algo que se assemelhará muito a construir um edifício, tendo que cuidar pessoalmente da colocação de cada tijolo, de cada prego, de cada janela, de cada porta, enfim, de cada pormenor.

Nada mais errado: a programação em *Assembly* é bastante produtiva, como irá reconhecer se continuar a seguir estes artigos.

Nesta família de computadores, o que torna o *Assembly* tão produtivo é que nenhuma outra linguagem consegue fazer um uso tão eficiente das centenas de rotinas já feitas e prontas a ser usadas e que se encontram à espera de serem invocadas dentro do nosso computador.

Um grupo dessas rotinas faz parte do BIOS (*Basic Input Output System*) e reside fisicamente no ROM do computador. O outro grupo vem com o sistema operativo MS-DOS. Tanto umas como outras estão amplamente documentadas em muitos livros.

Ainda em relação ao nosso mini-programa acima note o modo airoso como este é terminado:

“A aprendizagem das instruções de *Assembly* é fácil pois as mnemónicas estão organizadas por classes de instruções de código máquina de um modo que faz sentido para nós humanos.”

```
MOV AH,4C
INT 21
```

São 4 bytes e o DOS encarrega-se de tudo quanto é necessário para arrumar a casa antes de libertar a memória e lhe apresentar de novo o *prompt*. Podiamos ter terminado o programa de outros modos, por exemplo apenas com 2 bytes:

```
INT 20
```

O que no nosso caso serviria bem, pois não havia grandes arrumações a fazer, designadamente fechar outros programas abertos de dentro do nosso.

Tanto num caso como noutro invocamos rotinas do MS-DOS para executar o trabalho.

Contudo note que o *Assembly* não se substitui às linguagens de alto nível nem estas se substituem ao *Assembly*: cada qual tem o seu âmbito próprio e vamos ver qual é.

“As linguagens de alto nível não foram concebidas com o objectivo de dar resposta a toda e qualquer necessidade.”

QUANDO, ONDE E O QUE PROGRAMAR EM *ASSEMBLY* ?

De um modo geral, quanto mais se souber do computador com o qual se está a trabalhar, mais eficiente será o nosso trabalho de produzir programas.

As linguagens de alto nível não foram concebidas com o objectivo de dar resposta a toda e qualquer necessidade. Quando estivermos em dúvida devemos sempre seguir a velha regra que diz que, para qualquer problema específico, os meios ideais são os globalmente mais vantajosos.

Se conseguirmos resolver o problema a contento com uma linguagem como o BASIC, devemos fazê-lo sem qualquer hesitação. Ao fim e ao cabo é rápido escrever um programa em BASIC.

Uma instrução em BASIC corresponde normalmente a dezenas ou centenas de instruções em *Assembly*. Existem programas-ferramenta chamados «compiladores» que transformam as instruções que escrevemos em BASIC (referimo-nos naturalmente a versões de BASIC que suportam compilação), e que constituem o **código fonte**, em instruções de binário directamente inteligíveis pelo computador e ainda acrescentam tudo o que é necessário para o programa poder ser carregado em memória e executado. Porém, existem programas que não devem ser escritos, pelo menos integralmente em BASIC, e quem diz BASIC, diz PASCAL, ‘C’ ou qualquer linguagem de alto nível, designadamente:

1) Se existir necessidade de limitar o tamanho dos programas.

Isto, porque os compiladores nunca produzem código muito eficiente, existe muita redundância, muito código que nunca chega a ser executado.

Por exemplo, os programas destinados a ficar residentes em memória após passarem o controle para o DOS, e designados por TSR (do inglês *Terminate and Stay Resident*), deverão ser o mais curtos possíveis.

Numa linguagem de alto nível altamente compacta como o ´C´, é praticamente impossível construir um TSR, mesmo muito rudimentar, que ocupe para si um espaço de memória inferior a 15 KB, e este valor crescerá muito rapidamente se falarmos de PASCAL ou BASIC.

Em *Assembly* o mesmo TSR pode ocupar pouco mais de 256 bytes e não ocupa menos porque o MS-DOS reserva para si exactamente 256 bytes no ínicio de cada programa numa zona designada por **PSP** (*Program Segment Prefix*). Poderia eventualmente ocupar até menos de 256 bytes, mas isso obrigaria a entrar no terreno do **PSP**, pelo que o leitor fica desde já alertado que não é pratica recomendável.

2) Se for necessário que o programa seja rápido e atender a uma exigente temporização da sua execução.

Os programas de comunicações e todo o tipo de aplicações em que exista necessidade de apertado controle em tempo real.

3) Se houver necessidade de controlar o *hardware*, de dialogar directamente com os *chips*, de espreitar continuamente para dentro da memória e alterar os seus valores, de fazer uso directo das muitas rotinas que o DOS e o ROM-BIOS põem á disposição do programador.

E mesmo que para tal existam soluções nas linguagens de alto nível, deve-se ter em atenção que, se o programa que estiver a desenvolver se destinar a ser comercializado e a concorrência apresentar um programa curto e rápido que faça o mesmo que o seu longo e lento programa, isso pode causar-lhe várias dificuldades. E aqui surje a necessidade da programação em *Assembly*.

O *ASSEMBLY* É UTILIZADO EM DUAS VERTENTES:

1) Programas integralmente realizados nessa linguagem.

Normalmente são programas pequenos, raramente com mais de 10 KB.

Existem contudo excepções a essa regra, inclusivé programas com centenas de kilobytes.

2) Criação de rotinas ou módulos que podem ser ligadas com programas em linguagem de alto nível.

Normalmente essas rotinas executam acções críticas cuja adequada condução não pode ser realizada pelo reportório de instruções da linguagem de alto nível.

E, de facto, a importância da linguagem *Assembly*, ao contrário do que seria de esperar, é cada vez maior nos nossos dias. A maioria dos compiladores modernos possuem *interface* para ligação de módulos escritos em linguagem *Assembly* ,quer directamente quer após a sua passagem a **código objecto** (ver mais adiante a referência a **OBJ**).

E os compiladores mais recentes vão um passo mais adiante e permitem a incorporação directa de instruções em *Assembly* no seio de programas fonte escritos na linguagem de alto nível. Incluem-se neste grupo as recentes versões do Turbo Pascal e Turbo ‘C’.

E antes de continuar uma pequena nota:

Alguns autores classificam algumas linguagens, incluindo a linguagem ‘C’, por linguagem de «médio nível». Sem discutir a correcção do termo, com o qual pessoalmente concordamos, confirmamos que na realidade as linguagens de médio nível substituem em muitas situações o *Assembly*.

O sistema operativo MS-DOS e ambientes de lançamento de programas como o Windows foram, em larga escala, escritos em ‘C’. Contudo, essas linguagens de médio nível não substituiram integralmente a linguagem *Assembly*, e as instruções que possuem para execução a baixo nível são, quase sempre, mais complicadas que as mnemónicas de *Assembly*, e, curiosamente, não dispensam o seu conhecimento.

AS FERRAMENTAS PARA PROGRAMAR EM *ASSEMBLY*

Se a sua decisão neste momento for de que vale a pena aprender *Assembly*, então deverá ter ao seu dispor um

certo conjunto de ferramentas.

Nada de muito complicado nem dispendioso.

1) A ferramenta base designa-se por **Assemblador**, ou *Assembly* em inglês (os brasileiros utilizam o termo «montador»).

Mais do que entrar em análises semânticas, o importante aqui é saber que o Assemblador é para o *Assembly* o que o Compilador é para uma linguagem de alto nível.

Muita gente sabe que o MS-DOS é distribuído com um pequeno programa designado por DEBUG, mas poucos aprendem a trabalhar com ele, ou sabem sequer para que serve. Bem, o DEBUG é o programa mais poderoso dos utilitários que acompanham o MS-DOS, um autêntico prodígio de poder.

Já atrás falámos do DEBUG e vimos que contém um Assemblador, mas ficamos por aqui. O DEBUG embora valente, não passa de um «canivete suiço», uma ferramenta multiuso mas primitiva.

No campo dos verdadeiros Assembladores começamos por citar um que é de *shareware* e de nome A86 (que significa, segundo o seu autor, «Assemblador para a família de processadores 80x86 da Intel»).

É extremamente fácil aprender a trabalhar com o A86, pois ele dispensa quase totalmente o conhecimento prévio daquilo que se designa por «pseudo-operadores». Estes não são instruções *Assembly*, mas directivas para os Assembladores e são a causa número um de desânimo para muitos que tentam iniciar-se no estudo do *Assembly*.

Exemplo de pseudo-operadores são os célebres ASSUME, os PROC, os ENDP, etc.

E como o A86 demonstra muito bem, é possível construir programas algo sofisticados sem utilizar qualquer desses quebra-cabeças. Se os quiser utilizar muito bem, o A86 conhece-os também, senão, confie no bom senso do A86 que ele procede como se os pseudo-operadores lá estivessem e normalmente acerta.

O A86 tem muitos pontos a seu favor e, naturalmente,algumas pequenas limitações. Mas, em nossa opinião, um principiante terá toda vantagem em conhecer este programa-ferramenta, pois permitir-lhe-á um evoluir rápido para um conhecimento bastante elevado de linguagem *Assembly*. E, após esse conhecimento ser obtido, o tema dos pseudo-operadores torna-se muito mais fácil, quase intuitivo.

No âmbito dos Assembladores ditos profissionais devemos citar os mais conhecidos: O MASM e o TASM. O MASM é produto da Microsoft e considerado implicitamente a referência que qualquer dos outros Assembladores não pode perder de vista. Eles podem superar aqui e ali o MASM mas, acima de tudo, têm de poder afirmar que são 100% (ou quase) compatíveis com o MASM. Mesmo que não gostemos do MASM, devemos conhecê-lo, pois ele constitui a linguagem comum que todos temos de ter em qualquer área do conhecimento e o *Assembly* não é excepção.

O MASM foi, durante muitos anos, praticamente o único Assemblador profissional disponível no mercado.

O TASM surgiu em 1988, é um produto da Borland Internacional e, em nossa opinião, o melhor existente actualmente. Curiosamente, tanto o TASM como o MASM, nas versões mais recentes, têm aliviado bastante as suas exigências em matéria de directivas requeridas pelo Assemblador e aceitam uma forma simplificada dessas directivas baseada nos «modelos de memória» das linguagens de alto nível, forma essa que pode ser usada na grande maioria dos programas.

Os Assembladores actuam sobre o código *Assembly*, isto é, mnemónicas *Assembly* como as que vimos atrás e que escrevemos com um simples editor de texto como o EDIT do MS DOS 5.0, ou qualquer outro que produza texto não formatado, ou ASCII puro. Designa-se esse código por **programa fonte** (do inglês *source code*). Quando o MASM ou o TASM operam sobre esse programa, o resultado não é de imediato um programa executável com extensão COM ou EXE, mas sim um programa intermédio designado por **programa objecto** e que tem normalmente a extensão OBJ.

O A86, por seu lado, pode produzir também OBJ, mas por omissão construirá de imediato um executável COM, ou em alternativa um BIN, que é um programa binário puro que é carregado num endereço de memória de deslocamento zero. Devido à existência dos OBJ, é necessário termos outra ferramenta que é o ***linker***, ou ligador (ou «linkeditor» se o leitor for brasileiro).

2) O linker.

O MASM tem o seu *linker* de nome LINK, e o TASM tem o TLINK. Se tiver um OBJ produzido pelo A86, então peça «emprestado» um ou outro para **ligar** o programa. A existência de programas OBJ é uma «fatalidade» que tem a ver com:

– O programa pode ter referenciado rotinas ou variáveis de outros programas-módulo, ou funções que existem em bibliotecas de rotinas (*libraries* em inglês) e é necessário ligar tudo isso num único programa para que este possa funcionar.

– A estrutura de memória dos computadores baseados no 80x86 permite que os programas sejam carregados em qualquer ponto dessa memória para serem executados. Assim, outra função do *linker* é fabricar um cabeçalho (ou *Header* em inglês) que oriente o DOS no carregamento e relocação das várias partes do programa em memória. Deste modo, o resultado do trabalho do *linker* acaba por ser um programa com a extensão EXE, que congrega tudo o que é necessário para poder ser executado.

Mas alguns programas EXE têm a característica de não terem exigências particulares em matéria de relocação, pois a sua imagem em memória pode ser idêntica à sua imagem em ficheiro (*file* ou ficha são termos comuns, mas usaremos ficheiro por estar mais divulgado).

Num caso desses poderemos fazer uso de outra das ferramentas que é o:

3) EXE2BIN.

Este programa transforma (quando é possível) os ficheiros EXE em COM (ou noutras extensões mais arrevezadas como SYS, mas não compliquemos desnecessárimente nesta fase). Se dispuser do TLINK, o EXE2BIN não é necessário, pois basta usar o *switch* /t do TLINK para que o ficheiro venha com a extensão COM. Se o nome EX2BIN não lhe é estranho e nunca utilizou *Assembly*, talvez isso se deva a que algumas versões do MS-DOS trazem esse programa incluido. Porquê só o EXE2BIN acompanhar o MS-DOS, porquê às vezes o EXE2BIN e o LINK e outras vezes nem um nem outro, é um mistério não revelado pela Microsoft.

Que outras ferramentas existem ? Algumas outras, mas estas chegam para começar.

No artigo de hoje cobrimos um largo território para quem à partida muito pouco ou nada soubesse de *Assembly*. Mas toda a panorâmica que demos irá ajudá-lo muito nos próximos números, nos quais falaremos de arquitectura dos PC, segmentação de memória , interrupções, serviços do DOS e do BIOS e, simultâneamente, iremos aprofundando os nossos conhecimentos de *Assembly* sempre devidamente ilustrados com programas exemplo.

José Páscoa

“Mais do que entrar em análises semânticas, o importante aqui é saber que o Assemblador é para o *Assembly* o que o Compilador é para uma linguagem de alto nível.”