1 Introdução ao MSP430

25/01/2021

O MSP430 é um microcontrolador projetado e fabricado pela Texas Instruments com a intenção de abordar o mercado de sistemas embarcados e Internet das coisas (IoT). Ele disponibiliza uma grande quantidade de recursos de hardware e muitas opções de ultrabaixo consumo.

Sua arquitetura é denominada de “RISC-like”, o que significa que usa bastante dos recursos das arquiteturas RISC, mas não é uma máquina 100% RISC. Alguns recursos não RISC são usados, tais como instruções com diferentes tamanhos e tempo de execução, e a diversidade de instruções que podem acessar a memória.

1.1. O Que É um Microcontrolador ?

Para tornar bem claro o conceito de microcontrolador, consideremos o projeto de um sistema bem simples, por exemplo, o controlador para um elevador. No entanto, como ainda não temos a concepção de microcontrolador, proporemos o projeto com um microprocessador, desses clássicos usados para montar computadores. Como o controle de um elevador não pede grande capacidade de processamento, podemos optar por uma CPU (processador) mais simples, como um 8086 ou 80286 (antigos). Até mesmo um processador de 8 *bits*, como um 8085 ou Z-80, já seriam suficientes.

O que tal processador deveria controlar? Talvez uma série de itens: abrir e fechar a porta da cabine, controlar o tempo de abertura e fechamento dessa porta, ligar e desligar o motor, controlar sua direção de rotação (subir ou descer), receber os pedidos via teclado, monitorar a entrada e saída de pessoas através da interrupção de um feixe infravermelho junto à porta e até verificar o limite de carga do elevador. Com o intuito de fazer tudo isso, propomos a arquitetura da Figura 1.1.

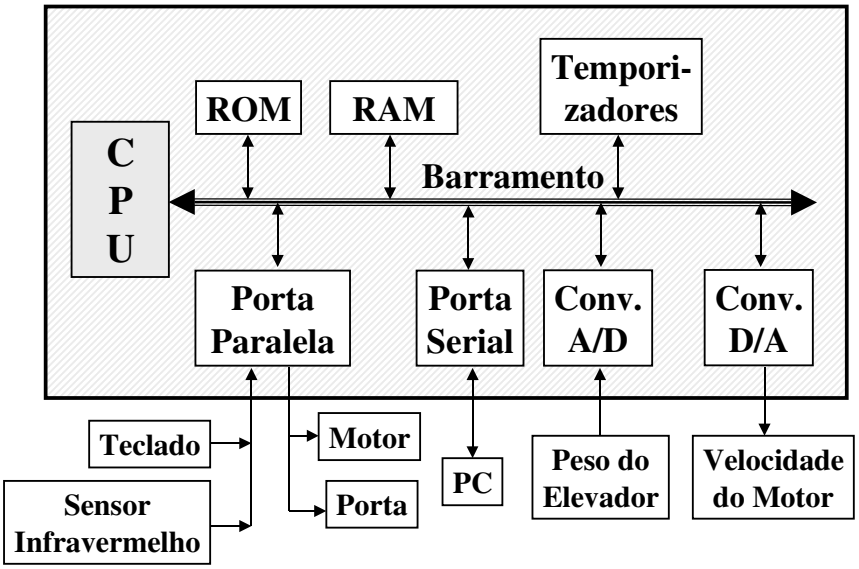

Figura 1.1. Arquitetura para o controlador de um elevador.

Ao observamos a arquitetura proposta, notamos uma série de blocos, que serão justificados a seguir. A memória ROM é usada para armazenar o programa de controle (*firmware*), de forma que, ao ser ligado, o sistema de *software* esteja sempre pronto. Além desta memória, é necessária uma memória RAM para armazenar as varáveis do programa e os resultados intermediários, ou seja, uma memória que sirva de rascunho. Para monitorar as entradas de teclado e de sensores infravermelhos usamos a porta paralela, onde cada bit corresponde à entrada da informação de cada um destes sensores. Para gerar as saídas que irão acionar o motor da porta e o motor principal, também nos valemos da porta paralela, sendo um bit para acionar cada motor. Essa porta paralela ainda serve para comandar o painel e acender luzes de alerta.

Para controlar o tempo em que a porta fica aberta ou fechada, a pausa entre a parada do elevador e a abertura da porta, além de uma série de outros intervalos, é necessário o emprego de temporizadores. Para ler o sensor que indica o peso do elevador, adicionamos um conversor analógico/digital (A/D). Usando a informação desse sensor, a CPU verifica se o elevador está sobrecarregado. A utilização do conversor digital/analógico (D/A) é justificada pelo fato do comando do motor principal ser controlado por tensão. Quanto maior a tensão, maior a velocidade de rotação, o que permite um

controle suave para a partida e a parada do elevador. Finalmente, para monitorarmos o funcionamento deste sistema de controle, conhecer seu estado e adquirir relatórios de falha, adicionamos uma porta serial. Assim, com um computador portátil externo seria possível fazer um diagnóstico completo do controlador proposto.

Portanto, para executar uma tarefa simples como a de controlar um elevador, chegamos facilmente a um sistema com um total de 6 ou 7 circuitos integrados (CI). A montagem de tal controlador exige uma placa de circuito impresso de tamanho razoável, envolve tempo no desenvolvimento do projeto eletrônico e demanda um custo de montagem que é proporcional ao número de componentes do sistema. Isso sem falar da probabilidade de falhas, que também é proporcional à quantidade de componentes.

Daí surge a pergunta: não é possível reunirmos tudo isso em um único circuito integrado? A resposta é sim. Mas que vantagens haveria ao colocar todos aqueles blocos da Figura 1.1 integrados dentro de um único *chip*?

- menor tamanho do controlador;
- menor consumo de energia;
- redução do tempo de projeto;
- redução do custo do projeto;
- maior confiabilidade e
- facilidade de manutenção.

Portanto, esta é uma ideia muito interessante. Mas, o que devemos esperar da CPU que será integrada dentro deste "controlador"? Será que ela precisa fazer contas em ponto-flutuante? Serão necessárias operações multimídia? É claro que não. Então, qual deve ser o perfil desta CPU?

- baixa capacidade de processamento;
- instruções compactas e velozes;
- capacidade de manipular bits;
- operação somente com inteiros;
- facilidade para adicionar-se *hardware*;
- facilidade de programação e
- flexibilidade na arquitetura de memória.

Chegamos assim a um circuito integrado que já traz incorporado todos os blocos apresentados na arquitetura da Figura 1.1. Uma arquitetura com estas características está claramente voltada para o controle de dispositivos e, por isso, recebe o nome de microcontrolador. De maneira bem simples podemos definir microcontrolador como um microprocessador simples, ao qual se incorporou uma grande quantidade de dispositivos com o intuito de usá-lo nas mais diversas aplicações de controle. O diagrama da Figura 1.2 apresenta o microcontrolador adequado para o controle do elevador.

Desta forma, ressaltamos agora a diferença entre microprocessador e microcontrolador. O microprocessador (µP) é dedicado ao processamento, oferece uma grande quantidade de

modos de acesso a dados, permite uma série de operações sobre esses dados e trabalha com operações em ponto-flutuante. Resumindo, serve para sistemas que fazem grandes quantidades de operações sobre os dados. Já com o microcontrolador (µC) a ideia é outra. Sua finalidade principal é o controle digital. Deve oferecer uma grande quantidade de recursos para entradas e saídas digitais, possibilidade de medir intervalos de tempo e viabilizar sistemas de pequeno tamanho físico. Ele não precisa realizar operações sofisticadas sobre os dados. Uma conclusão óbvia: nunca um microcontrolador será usado para construir um computador.

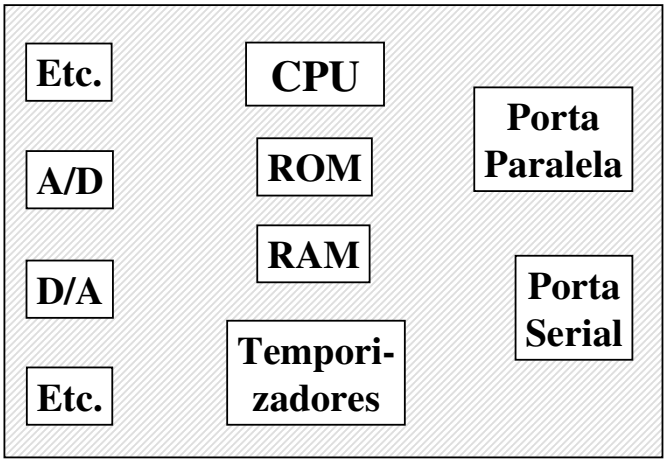

Figura 1.2. Um microcontrolador para controlar o elevador.

1.2. *Pipeline* do MSP 430

O MSP430 tem um *pipeline* de 3 estágios:

- (B) Busca: que acessa a memória e traz a próxima instrução a ser executada.
- (D) Decodifica: que "entende" o que faz a instrução e
- (E) Executa: que executa a instrução.

A Figura 1.3 ilustra esses três estágios, imaginando que a instrução de endereço 100 está sendo executada, que a instrução de endereço 101 está sendo decodificada e que a unidade de busca está carregando a instrução de endereço 102.

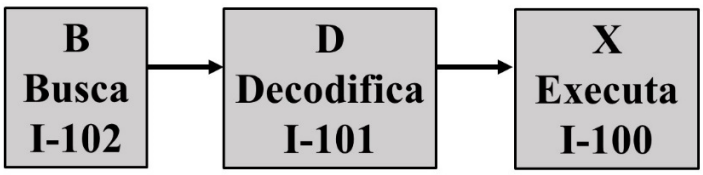

Figura 1.3. Pipeline em operação.

Neste modelo simplificado, considerando que o *pipeline* esteja em "ritmo" e que T seja o período do relógio do *pipeline*, a execução de cada instrução (B – D – X) consome 3 períodos, entretanto, na saída temos uma instrução executada a cada período. Isso é possível porque temos 3 instruções sendo operadas simultaneamente pelo *pipeline*. Assim taxa de execução (*throughput*) de instruções é de 1/T.

Note que no início, a primeira instrução que entra no *pipeline* demora 3T para ser executada, a segunda vai demorar 2T e depois o *pipeline* entra em ritmo e entrega uma instrução executada a cada período. As instruções de salto podem quebrar o ritmo desse *pipeline*. Por exemplo, se a instrução de endereço 100 for um salto para a o endereço 500, o trabalho antecipado feito pela decodificação da instrução de endereço 101 e a busca pela instrução de endereço 102 serão perdidos.

Para se antecipar a este problema, a Unidade de Busca possui inteligência para entender que buscou uma instrução de salto e assim orientar as próximas buscas para este novo endereço. O problema maior surge quando a instrução é um salto condicional. Por exemplo, saltar para o endereço 500 caso uma operação resulte em zero. Repare que no momento da busca, a operação ainda não foi executada e por isso a unidade de busca não tem como decidir se o salto vai ou não acontecer. Vejamos este caso com mais detalhe usando o programa listado a seguir.

```
100 MOV     #20,R5
101 ADD     R5,R8
102 JZ      500
103 MOV     #1,R9 ;R5+R8 ≠ 0
...
500 MOV     #2,R9 ;R5+R8 = 0
...
```

Neste caso, quando a Unidade de Busca carrega a instrução de endereço 102, ela vê que se trata de um salto condicional: `JZ 500` (saltar se o resultado da operação anterior resultou em zero), porém, a operação `ADD R5,R8` ainda está na Unidade de Decodificação. A Unidade de Busca está então com um dilema: a próxima instrução será a 103 ou a 500. Se ela errar, o *pipeline* deverá ser esvaziado e vai gastar 3 períodos para entrar em ritmo.

A solução adotada no projeto do MSP foi a de dar mais recurso para a Unidade de Busca, que passa a trabalhar de forma especulativa. Para não decidir errado, ela carrega as duas próximas possíveis instruções: a de endereço 103 e a de endereço 500. Na tabela abaixo, note que no instante T_n a conta é feita. Assim, quando precisar enviar adiante (para a

decodificação) uma das duas instruções (instante Tn+1), a unidade de Busca sabe se o resultado deu zero, ou não, e envia a instrução correta.

Tabela 1.1. Ilustração da Busca Especulativa no instante T_n

	T_{n-1}	T_n	T_{n+1} (R5+R8≠0)	T_{n+1} (R5+R8=0)
B	`102 JZ 500`	`103 MOV #1,R9` `500 MOV #2,R9`	`104 ...`	`501 ...`
D	`101 ADD R5,R8`	`102 JZ 500`	**`103 MOV #1,R9`**	**`500 MOV #2,R9`**
X	`100 MOV #20,R5`	`101 ADD R5,R8`	`102 JZ 500`	`102 JZ 500`

1.3. Registradores

Denomina-se "Arquitetura do Programador" os recursos e particularidades que o programador tem à sua disposição para operar um processador. Uma porção muito importante são os registradores.

O MSP430 é um processador de 16 bits, o que significa que sua ALU (Unidade Lógica e Aritmética) opera com palavras de 16 bits. O programador tem acesso a 16 registradores, denominados R0, R1, ..., R15, todos de 16 bits. Na verdade, nem todos esses registradores estão realmente disponíveis. A Figura 1.4 apresenta um resumo desses registradores.

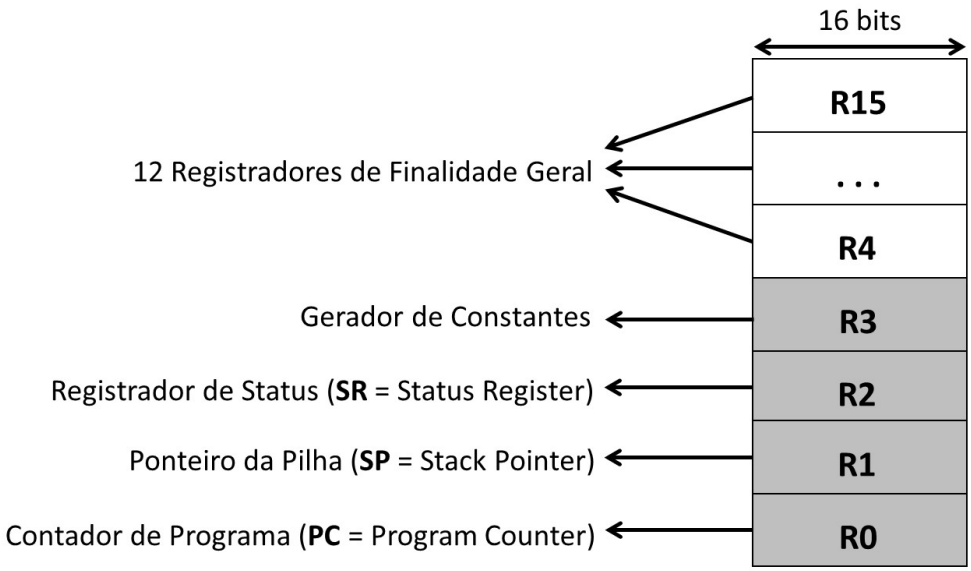

Figura 1.4. Registradores do MSP430.

R0 = PC = Contador de Programa → A CPU usa este registrador para o sequenciamento da execução das instruções. Durante a execução de uma instrução o PC sempre aponta para o endereço da próxima instrução.

R1 = SP = Ponteiro da Pilha → Este registrador é usado para criar a pilha, onde são armazenados os endereços de regresso de sub-rotinas. A pilha também pode ser usada para armazenamento temporário.

R2 = SR = Registrador de *Status* → Ele fornece informações sobre a última operação realizada pela ALU, além de controlar acesso aos modos de baixo consumo de energia. Dependendo da forma de endereçamento (ao ser lido), este registrador fornece as constantes 0, +4, +8. Mais adiante veremos detalhes sobre essa geração de constantes.

R3 = Gerador de Constantes → Dependendo da forma de endereçamento (ao ser lido), este registrador fornece as constantes 0, +1, +2, -1. Mais adiante veremos detalhes sobre essa geração de constantes.

R4, R5, ..., R15 = Registradores de Finalidade Geral → O programador tem liberdade plena para usar esses 12 registradores.

O Registrador de Status (R2 = SR) merece uma abordagem mais cuidadosa. A Figura 1.5 apresenta o detalhamento dos bits deste registrador. Por hora, temos 4 bits importantes, denominados V, N, Z e C. Esses bits são, de forma simples, denominados *flags*. Eles fornecem informações sobre o resultado da última operação realizada pela ALU. Em outras palavras, eles mudam de valor a cada operação da ALU.

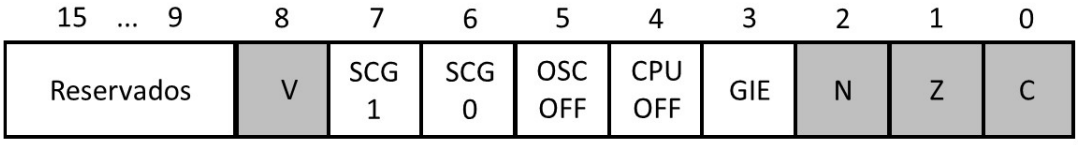

Figura 1.5. Detalhamento do Registrador de Status (R2 = SR).

C = Carry → Este bit (*flag*) é o "vai um". Ele indica se o resultado da operação aritmética foi muito grande para caber no espaço destinado. Em outras palavras, é o bit de transbordamento (*overflow*).

Z = Zero → Este bit (*flag*) indica se o resultado da última operação foi igual a zero. Cuidado, se Z = 1, é porque o resultado da operação foi igual a zero. Se Z = 0, é porque o resultado da operação foi diferente de zero.

N = Negativo → Este bit (*flag*) indica se o resultado da operação foi um número negativo (N = 1) ou positivo (N = 0). Ele, simplesmente, é igual ao bit mais significativo do resultado da operação realizada.

V = Overflow (complemento a 2) → Este bit (*flag*) indica se o resultado da última operação não coube no espaço a ele destinado. Ele é usado nas operações com sinal (complemento a 2).

1.2. Memória

O MSP possui uma única memória para programas e dados. É o que se denomina de arquitetura de von Neumann. Existe também a arquitetura de Harvard, que separa a memória de programa da memória de dados.

Cada versão do MSP430 tem sua particular arrumação de memória. Vamos nos concentrar na memória do MSP430F5529, que está apresentada na Figura 1.6. Faremos a seguir a descrição de cada um desses segmentos. Mas antes é necessária uma pequena diferenciação entre os termos Flash e RAM.

Flash (EEPROM) → Memória não volátil. O que é escrito numa memória Flash se mantém, mesmo na ausência de energia.

RAM (SRAM) → Memória volátil. O conteúdo desta memória é perdido na ausência de energia.

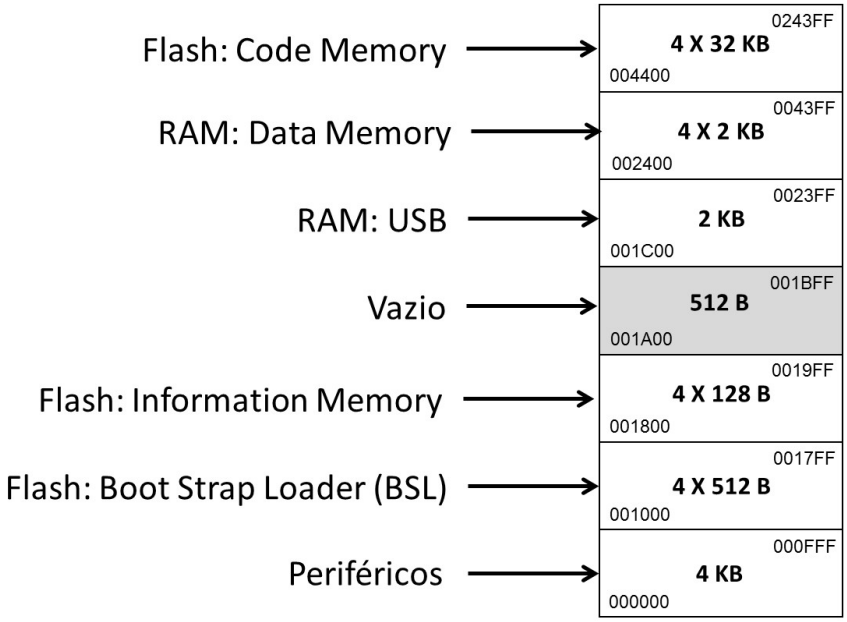

Figura 1.6. Mapa de memória do MSP430 (F5529).

Code Memory (Flash com 128 KB, de 0x4400 até 0x243FF), é a memória onde é carregado o programa do usuário. Ela está organizada em 4 bancos de 32 KB. Por ser uma memória não volátil, o programa carregado se mantém, mesmo na ausência de energia. Como este curso não faz uso do endereçamento estendido, ficaremos limitados à faixa de 0x4400 até 0xFFFF (47 KB). Dentro desta área, na faixa de 0xFF80 até 0xFFFF é construída a Tabela de Vetores de Interrupção. O estudo sobre interrupção é tema de um capítulo mais adiante. Todo o programa de usuário inicia em 0x4400.

Data Memory (RAM com 8 KB, de 0x2400 até 0x43FF), é a memória usada para armazenar os dados do usuário. Ela está organizada em 4 setores de 2 KB, sendo que cada setor pode ser desabilitado para economizar energia. Por ser uma memória volátil, os dados são perdidos na ausência de energia. As variáveis de usuário são criadas a partir de 0x2400.

USB (RAM com 2 KB, de 0x1C00 até 0x23FF), é a memória reservada para o USB. Se não se usar USB, ela serve como memória genérica. Por ser uma memória volátil, os dados são perdidos na ausência de energia.

Vazio (512 B, de 0x1A00 até 0x1BFF), é um espaço de endereços onde não existe memória.

Information Memory (Flash de 512 B, de 0x1800 até 0x19FF), é uma memória destinada a armazenar informações não voláteis. Ela está organizada em 4 segmentos de 128 B. O usuário pode armazenar nesta área dados que devem ser consultados toda a vez que o processador for energizado. Por exemplo, dados sobre a última configuração usada. Cada segmento pode ser protegido contra alterações (contra escrita ou apagamento).

Boot Strap Loader (Flash com 2 KB, de 0x1000 até 0x17FF), é uma memória que contém códigos para a inicialização do CPU e a comunicação serial com o computador. Usada para receber o programa do usuário e também para depuração (*debug*) de programas.

Periféricos (RAM com 4 KB, de 0x0000 até 0x0FFF), é a memória usada para mapear todos os registradores que controlam os recursos internos. O MSP não possui espaço de endereço para I/O (entrada/saída). Assim, os dispositivos de I/O são mapeados nesta área de memória.

Como pode ser notado, as memórias são medidas em bytes, entretanto, o MSP430 é um processador de 16 bits. Vamos ver como funciona a organização de memória de forma a permitir acessos de 8 bits e de 16 bits.

A memória, afinal de contas, é arrumada em bytes. É claro que palavras de 16 bits usam 2 bytes. A pergunta é: para armazenar uma palavra de 16 bits, qual byte armazenamos primeiro, o mais significativo (MSB) ou o menos significativo (LSB)? Existem duas óbvias possibilidades, como mostrado na Figura 1.7.

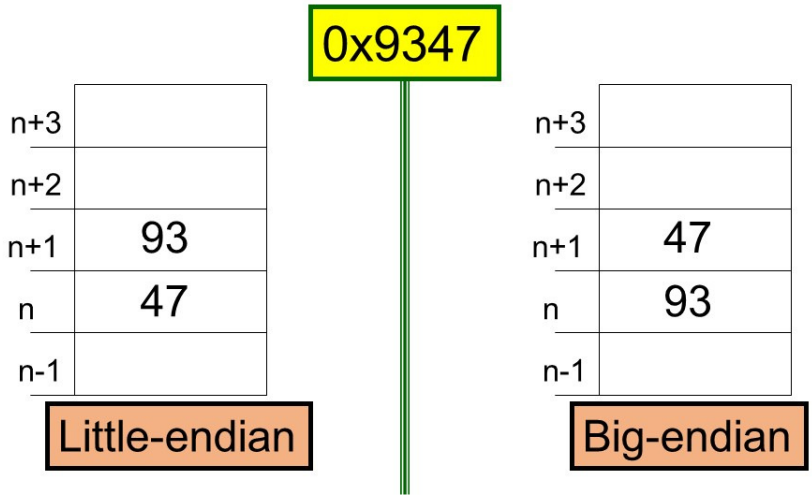

Figura 1.7. Duas possibilidades para o armazenamento dos bytes que compõe a palavra de 16 bits 0x9347.

Little-endian → no sentido crescente dos endereços, armazena-se primeiro o byte menos significativo (LSB) e depois, é claro, o byte mais significativo (MSB).

Big-endian → no sentido crescente dos endereços, armazena-se primeiro o byte mais significativo (MSB) e depois, é claro, o byte menos significativo (LSB).

A título de exemplo, apresentamos uma tabela hipotética com o conteúdo em hexadecimal da memória RAM a partir do endereço 0x2400.

Tabela 1.2. Exemplo de conteúdo da memória RAM do MSP

2400	2401	2402	2403	2404	2405	2406	2407
01	23	45	67	89	AB	CD	EF

Segundo as duas organizações possíveis, temos as seguintes palavras de 16 bits, apresentadas na Tabela 1.3:

Tabela 1.3. Duas possibilidades de organização em 16 bits do conteúdo de memória apresentado na Tabela 1.1

	Palavra de 16 bits	
Endereço	*Little-endian*	*Big-endian*
2400	2301	0123
2402	6745	4567
2404	AB89	89AB
2406	EFCD	CDEF

Não existe qualquer vantagem de uma organização sobre a outra. Para qualquer uma delas, o endereço da palavra de 16 bits é o endereço do primeiro byte que foi armazenado. O MSP430 segue a organização *Little-endian*.

Para permitir que a memória do MSP possa ser acessada em 8 bits e em 16 bits é usada uma arquitetura semelhante à que está apresentada na Figura 1.8. Para esta explicação, foram considerados apenas os primeiros 64 KB endereços (desde 0x0000 até 0xFFFF) com as 16 linhas de endereços numeradas de A0 até A15. É preciso notar que existem os dois bancos de memória de 8 bits: um banco com os endereços pares e outro com os endereços ímpares. Um endereço par tem A0 = 0 e um endereço ímpar tem A0 = 1.

A habilitação de cada banco é controlada pelo pino #CE (*Chip Enable*), ativo em nível baixo, ou seja, o banco de memória funciona com #CE = 0. Para o acesso a bytes, o sinal #BHE é o inverso da linha A0.

O acesso (leitura ou escrita) a um byte com endereço par (A0 = 0) habilita o #CE da memória par. A memória ímpar permanece desabilitada pois o #BHE = 1, já que ele é o inverso de A0.

O acesso (leitura ou escrita) a um byte com endereço ímpar (A0 = 1) habilita o #CE da memória ímpar, pois #BHE = 0. A memória par permanece desabilitada pois o A0 = 1.

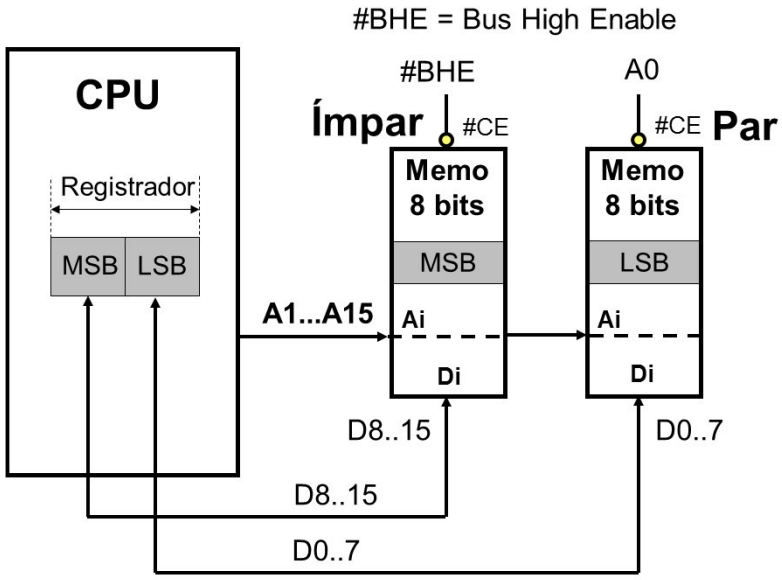

Figura 1.8. Arquitetura de memória do MSP430 (F5529).

Para o acesso (leitura ou escrita) a uma palavra de 16 bits em endereço par, os dois bancos de memória terão de ser habilitados ao mesmo tempo. Se o endereço é par (A0 = 0) o banco par já é habilitado automaticamente e além disso, como é um acesso de 16 bits, a CPU faz #BHE = 0, habilitando assim o banco ímpar. Em outras palavras, os dois bancos são habilitados ao mesmo tempo e as memórias entregam ou recebem os dois bytes que compõem a palavra de 16 bits.

É importante notar que as palavras de 16 bits em endereços pares têm seus bytes "alinhados" nos dois bancos de memória. O valor da linha A0 é a única diferença entre o endereço do LSB e do MSB desta palavra. Por isso, o manual do MSP430 indica que palavras de 16 bits devem ser armazenadas em endereços pares.

Surge uma pergunta: o que acontece se tentarmos acessar uma palavra de 16 bits em um endereço ímpar? Por exemplo, considerando a Tabela 1.2, qual o resultado do acesso em 16 bits ao endereço 0x2401? O manual do MSP é omisso a esse respeito. O leitor é convidado a fazer este teste. Será comprovado que o último bit de endereço é ignorado, ou seja, serão usadas as posições 0x2400 e 0x2401.

Alerta ao usuário que já conhece um pouco de assembly.
Veja a reserva de espaços apresentada abaixo, que aloca duas variáveis.3 variáveis.

```
      MOV.B       #0X12,&VAR1
      MOV.W       #0X3456,&VAR1
      JMP         $
      .data
VAR1: .space      1
```

```
VAR2: .space      2
```

A área de dados do usuário tem início em 0x2400. Assim temos os seguintes endereços:
VAR1 = 0x2400
VAR2 = 0x2401
Quando se escrever uma palavrar de 16 bits em VAR2, o valor de VAR1 será alterado. O resultado será &VAR1 = 0x56.