PROCESSADOR MULTI-CICLO – R12

1 CARACTERÍSTICAS GERAIS DAS ARQUITETURAS RX

- As arquiteturas Rx são do tipo *load-store*, ou seja, as operações lógicas e aritméticas são executadas exclusivamente entre registradores da arquitetura ou entre constantes imediatas e registradores. As operações de acesso à memória só executam ou uma leitura (*load*) ou uma escrita (*store*).
- Devido à característica *load-store*, o processador deve disponibilizar um conjunto relativamente grande de registradores, para reduzir o número de acessos à memória externa, pois estes representam perda de desempenho em relação a operações entre registradores internos ao processador. Esta característica difere da arquitetura baseada em acumulador, a qual mantém todos os dados em memória, realizando as operações aritméticas entre um conteúdo que está em memória e um, ou poucos registradores especiais, denominados acumuladores. Considere o exemplo: `for(i=0; i<1000; i++)`. Neste exemplo, caso `i` esteja armazenado em memória, tem-se 2000 acessos à memória, realizando leitura e escrita a cada iteração do laço `for`. Caso tenha-se o valor de `i` armazenado em um registrador interno, apenas opera-se sobre este, sem acesso à memória externa durante a maior parte do tempo. Considerando-se que o tempo de acesso a um registrador é normalmente uma ordem de grandeza (10 vezes) menor que o tempo de acesso a uma posição de memória, percebe-se o ganho que se pode auferir em relação a arquiteturas baseadas em acumulador.
- Dados e endereços nas arquiteturas Rx são de 16 bits. Logo, diz-se que a **palavra** destes processadores é de 16 bits, ou que se trata de processadores de 16 bits.
- O endereçamento de memória é orientado a palavra, ou seja, cada endereço corresponde a um identificador de posição onde residem 16 bits o que se denomina uma palavra de conteúdo.
- O banco de registradores possui 16 registradores de uso geral, de 16 bits cada um.
- Há um formato regular para as instruções, todas possuem exatamente o mesmo tamanho, e ocupam 1 palavra de memória. Deve-se comparar esta situação com a que ocorre na arquitetura Cleópatra. A instrução contém o código da operação e o(s) operando(s), caso exista(m).

Assim, este processador é praticamente uma máquina RISC, faltando contudo algumas características que existem em qualquer RISC, tal como *pipelines*, assunto que será introduzido ao final desta disciplina e estudado em profundidade maior em Arquitetura de Computadores I.

2 CARACTERÍSTICAS ESPECÍFICAS DO PROCESSADOR R12

- Instruções na arquitetura R12 têm duração variável entre 2 e 5 ciclos de relógio. Logo, o CPI de qualquer organização não pipeline para esta arquitetura possui CPI entre 2 e 5. Isto deve ser comparado com a arquitetura Cleópatra.
- Existem quatro qualificadores de estado (*flags*) na R12. Estes qualificadores são: o sinal do resultado de uma operação em complemento de 2 (*flag* `N`, quando igual a 1, indica sinal negativo); o fato de uma operação resultar no valor 0 (*flag* `Z`, quando igual a 1, indica que o resultado foi 0); o fato de uma operação aritmética resultar em um vai-um (*flag* `C`, quando igual a 1, indica que houve vai-um); e finalmente o fato de uma operação aritmética ultrapassar a capacidade do processador de representar o resultado da operação entre números inteiros corretamente, ou seja transbordo de campo (*flag* `V`, quando igual a 1, indica que houve transbordo).
- Modos de endereçamento que implicam múltiplos acessos à memória (tais como o modo indireto) não existem. O modo direto ou absoluto é limitado a casos especiais, em que seu uso é inevitável e produz código relocável. Os modos a registrador, relativo e suas combinações são privilegiados na R12.
- O processador possui suporte em hardware para a implementação de pilhas, através do registrador de controle apontador de pilha (em inglês, *stack pointer*), e de instruções específicas para controlar este. Pode-se usar este suporte para a implementação de estruturas de dados do tipo pilha em memória, úteis e. g. para salvamento de contexto durante a chamada e o retorno de subrotinas.

3 A RELAÇÃO PROCESSADOR - MEMÓRIAS – AMBIENTE EXTERNO

A Figura 1(a) mostra como deve ser implementada a organização do processador R12 neste trabalho. Nela, está explicitada a relação entre o processador, as memórias externas e o mundo exterior ao subsistema processador – memórias de dados e instruções, qual seja os responsáveis pela geração dos sinais de clock e reset. Uma diferença primordial entre esta organização e algumas organizações propostas em trabalhos anteriores e o processador Celópatra é que a R12 deve ser implementada como uma **arquitetura Harvard**, ou seja, o processador deve usar interfaces distintas para as memórias de instruções e de dados. Alguns trabalhos anteriores previam o uso de uma interface de memória unificada para instruções e dados, caracterizando uma organização do tipo von Neumann. Segue agora uma breve discussão da organização R12 a ser implementada.

O processador R12 recebe do mundo externo dois sinais de controle. O **clock** é o primeiro destes, que sincroniza todos os eventos internos do processador. O segundo sinal, denominado **reset**, leva o processador a reiniciar a execução de instruções a partir do endereço 0000H da memória. O processador ainda fornece ao mundo externo informações sobre seu estado interno através de dois sinais. O sinal **exception** indica se o processador está operando normalmente (quando **exception** =0), ou se alguma condição de exceção foi encontrada (quando **exception** =1). Quando existe alguma condição de exceção sinalizada pelo sinal **exception**, o sinal **cause** dá o código desta condição. Pelo menos duas condições devem ser identificadas: (1) processador parado devido à execução de uma instrução **HALT**, (2) processador parado devido à tentativa de execução de uma instrução inválida. Estas exceções provocam uma parada (natural e forçada, respectivamente) do processador. No caso de parada do processador devido a uma exceção, apenas o sinal externo **reset** pode provocar o reinício da execução de instruções.

Os sinais providos pelo processador R12 para a troca de informações com as memórias são:

- **i_address** – um barramento unidirecional de 16 bits que define sempre o endereço da posição de memória contendo a instrução a ser buscada a seguir (ou, transitoriamente, que acabou de ser buscada);
- **instruction** – um barramento unidirecional de 16 bits, apresentando a instrução contida na posição de memória dada por **i_address**;
- **d_address** – um barramento unidirecional de 16 bits, contendo o endereço da posição de memória a ser acessada para leitura ou escrita de dados, da ou para a memória de dados, respectivamente;
- **data** - um barramento bidirecional de 16 bits transportando dados do ou para o processador R12;

Ambas interfaces de memória são assíncronas, ou seja, não dependem de sinais de relógio tais como o sinal **clock**. Não existem sinais de controle para acesso à memória de instruções, pois não há fluxo bidirecional de informação. A memória de instruções é vista pelo R12 como uma memória de apenas leitura, que fornece informações na sua saída (instruções) a partir do estabelecimento do endereço de memória pelo processador no barramento **i_address**. O controle de acesso à memória de dados é feito pelo processador através dos sinais **ce** e **rw**. O sinal **ce** indica se está em curso uma operação com a memória de dados (quando **ce** = 1) e o sinal **rw** indica se esta operação é de escrita (quando **rw** = 0) ou de leitura (quando **rw** = 1). Obviamente, quando **ce** = 0 o valor do sinal **rw** é irrelevante, a exemplo do que ocorria na Cleópatra.

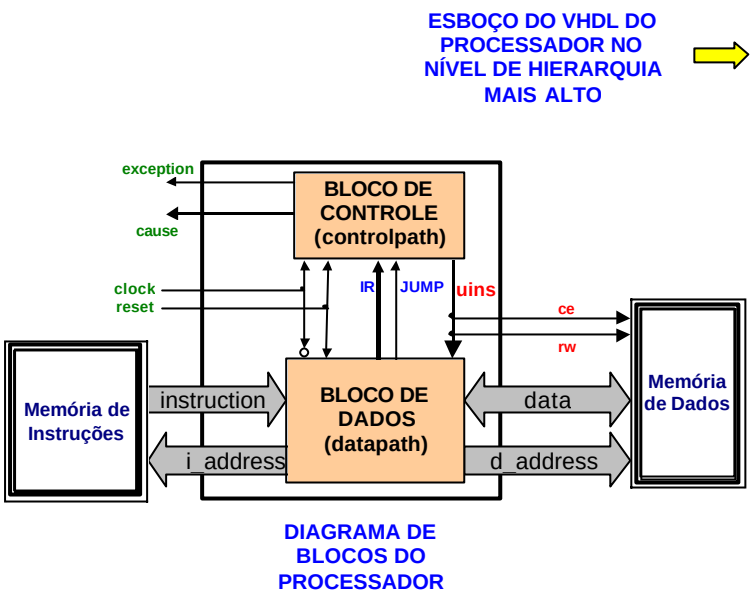
O bloco de controle gera a palavra de microinstrução (**uins**) para comandar a execução passo a passo das instruções pelo bloco de dados. A microinstrução é responsável por especificar cada uma das ações unitárias que serão executadas pelo hardware do bloco de dados a cada ciclo de relógio, ou seja as microoperações. Exemplos destas são cada uma das três seleções de registradores a serem escrito/lidos no/do banco de registradores, a operação que a ALU (do inglês, *Arithmetic Logic Unit*) executará, e os sinais de controle de acesso à memória de dados externa.

O bloco de dados envia para o bloco de controle a instrução corrente (conteúdo do registrador **IR**). O bloco de dados também é responsável pela troca de informações com a memória externa.

É importante ressaltar que os blocos de dados e de controle operam sempre em fases distintas do sinal **clock**. Na borda de subida de **clock** o bloco de controle gera a microinstrução, e na borda de descida o bloco de dados executa esta, modificando seus registradores internos. Com isto, sempre tem-se dados estáveis nas transições de **clock** em cada um dos blocos[1].

A Figura 1(b) representa o nível mais alto da hierarquia do processador, através da linguagem VHDL. Nesta Figura, deve-se notar que os blocos de dados e de controle são instanciados e estão conectados entre si por *sinais*, conforme definido pelos comandos *port map* no processo de instanciação.

[1] Deve estar claro que esta última afirmação é verdade apenas se a freqüência de relógio for suficientemente baixa para permitir que os sinais estabilizem em no máximo meio período de relógio (entre duas bordas opostas consecutivas do sinal **clock**).



(a) Diagrama de blocos processador-memória para a organização R12.

```
entity R12 is
    port( clock,reset: in std_logic;
          ce,rw,exception: out std_logic;
          cause: out R12_exception;
          i_address, d_address: out reg16;
          instruction: in reg16;
          data: inout reg16);
end R12;
architecture R12 of R12 is
      signal IR: reg16;
      signal jump: std_logic;
      signal uins: microinstrucao;
 begin

     dp: entity work.datapath
         port map(ck=>clock, rst=>reset,
           IR=>IR, jump=>jump,uins=>uins,
           i_address=>i_address,
           instruction=>instruction,
           d_address=>d_address, data=>data);

     ct: entity work.controlpath
         port map(ck=>clock, rst=>reset,
          exception=>exception, cause=>cause,
          IR=>IR, jump=>jump, uins=>uins);

      ce <= uins.ce;
      rw <= uins.rw;
end R12;
```

(b) Descrição VHDL do nível mais alto da hierarquia de descrição do processador R12.

Figura 1 - Relação entre o processador, ambiente externo, memórias externas e esboço de descrição VHDL de topo para a organização R12 a ser implementada.

4 CONJUNTO DE INSTRUÇÕES

A Tabela 1 descreve resumidamente cada instrução da arquitetura R12, usando as convenções a seguir.

Convenções Utilizadas na Tabela 1:

- **Rt (*target register*)** é o registrador usado na maioria das instruções como destino dos dados processados;
- **Rs1** e **Rs2** são registradores usados na maioria das instruções como origem dos operandos para obter os dados;
- O sinal ← é usado para designar atribuição (escrita) de valores resultantes da avaliação da expressão à direita do sinal ao registrador ou posição de memória identificada à esquerda do sinal;
- Os identificadores **imed4**, **imed8** e **imed12** representam operandos imediatos de 4, 8 e 12 bits, respectivamente;
- As expressões **Rt_high** e **Rt_low** representam as metades (8 bits) superior e inferior do registrador **Rt**;
- O operador **&** representa a concatenação de vetores de bits;
- A expressão **PMEMD(X)** representa o conteúdo de uma posição de memória de dados cujo endereço é **X** (na leitura) ou a própria posição de memória de dados (na escrita);
- Está implícito em todas as instruções o incremento do registrador PC após a busca da instrução. Qualquer outra referência a manipulação do PC é parte da semântica da instrução particular;
- **Extensão de sinal** é a operação que transforma um dado vetor de bits em outro maior, mas cujo valor em complemento de 2 é equivalente. Consiste em copiar o bit de sinal (ou seja, o bit mais significativo do vetor) à esquerda do vetor original tantas vezes quanto seja necessário para gerar o vetor maior. Por exemplo, na instrução **JPMI**, se **imed12** for 1111 1111 1111 (-1 em complemento de 2, 12 bits), a extensão de sinal transforma este vetor em 1111111111111111 (-1 em complemento de 2, 16 bits). A operação é trivialmente correta também para números positivos, onde o bit de sinal estendido é o bit 0. Quando se menciona extensão de sinal, os valores imediatos representam números em complemento de 2. Caso contrário, os números são representações em binário puro, o que ocorre, por exemplo, nas instruções **ADDI** e **SUBI**;
- Cada instrução pode ou não afetar os qualificadores de estado **N, Z, C, V**. O fato de uma instrução afetar um ou mais qualificadores está explicitamente indicado no campo Ação da **Tabela 1**, acrescentando ao final da descrição da semântica da instrução uma lista entre parênteses indicando os flags afetados, quando estes existirem;
- As instruções de deslocamento de bits sempre operam sobre um bit de cada vez. Estas inserem 0s no extremo oposto ao sentido do deslocamento.
- Todas as instruções de controle de fluxo, com exceção da instrução de retorno de subrotina (**RTS**) e da instrução de parada do processador (**HALT**) são condicionais. Isto compreende todos os saltos e chamadas de subrotina. Um salto/chamada de subrotina somente é executado se uma função Booleana, denominada **JUMP**, avaliar como verdadeira (valor lógico 1). Mais detalhes sobre o funcionamento de saltos serão apresentados na Seção 4.3.

Tabela 1 – Mnemônicos, codificação e semântica resumida das instruções do processador R12.

Instrução	FORMATO DA INSTRUÇÃO				AÇÃO
	15 – 12	11 - 8	7 - 4	3 - 0	
ADD Rt, Rs1, Rs2	0	Rt	Rs1	Rs2	Rt ← Rs1 + Rs2; *(todos os flags)*
SUB Rt, Rs1, Rs2	1	Rt	Rs1	Rs2	Rt ← Rs1 - Rs2; *(todos os flags)*
AND Rt, Rs1, Rs2	2	Rt	Rs1	Rs2	Rt ← Rs1 and Rs2; *(N e Z)*
OR Rt, Rs1, Rs2	3	Rt	Rs1	Rs2	Rt ← Rs1 or Rs2; *(N e Z)*
XOR Rt, Rs1, Rs2	4	Rt	Rs1	Rs2	Rt ← Rs1 xor Rs2; *(N e Z))*
ADDI Rt, imed8	5	Rt	Imed8		Rt ← Rt + ("00000000" & Imed8); *(todos os flags)*
SUBI Rt, imed8	6	Rt	Imed8		Rt ← Rt - ("00000000" & Imed8); *(todos os flags)*
LDLI Rt, imed8	7	Rt	Imed8		Rt ← Rt_high & imed8; *(N e Z)*
LDHI Rt, imed8	8	Rt	Imed8		Rt ← imed8 & Rt_low; *(N e Z)*
LDIM Rt, Rs2, Imed4	9	Rt	Imed4	Rs2	Rt ← PMEMD ("000000000000" & Imed4 + Rs2)
LD Rt, Rs1, Rs2	A	Rt	Rs1	Rs2	Rt ← PMEMD (Rs1+Rs2)
ST Rt, Rs1	B	Rt	Rs1	0	PMEMD (Rt) ← Rs1
COMP Rt, Rs1	C	Rt	Rs1	0	Rt – Rs1; *(todos os flags)*
NOT Rt, Rs1	C	Rt	Rs1	1	Rt ← not (Rs1); *(N e Z)*
SL Rt, Rs1	C	Rt	Rs1	2	Rt ← Rs1 deslocado 1 bit à esq, com 0s à dir.; *(N e Z)*
SR Rt, Rs1	C	Rt	Rs1	3	Rt ← Rs1 deslocado 1 bit à dir, com 0s à esq.; *(N e Z)*
MOV Rt, Rs1	C	Rt	Rs1	4	Rt ← Rs1; *(N e Z)*
MOVSP Rt	C	Rt	0	5	Rt ← SP
STMSK imed8	C	Imed8		A	MASK ← Imed8
JPRGR Rs1	C	0	Rs1	B	Se JUMP=1, PC ← PC + Rs1
JSRGR Rs1	C	0	Rs1	C	Se JUMP=1, SP ← SP-1; PMEMD(SP-1) ← PC; PC ← PC + Rs1
JPRG Rs1	C	0	Rs1	D	Se JUMP=1, PC ← Rs1
JSRG Rs1	C	0	Rs1	E	Se JUMP=1, SP ← SP-1; PMEMD(SP-1) ← PC; PC ← Rs1
JPMI imed12	D	Imed12			Se JUMP=1, PC ← PC + (Imed12 com ext. de sinal)
JSRMI imed12	E	Imed12			Se JUMP=1, SP ← SP-1; PMEMD(SP-1) ← PC; PC ← PC + (Imed12 com ext. de sinal)
POP Rt	F	Rt	0	0	Rt ← PMEMD(SP); SP ← SP+1
PUSH Rs1	F	0	Rs1	1	SP ← SP-1; PMEMD(SP-1) ← Rs1
LDSP Rs1	F	0	Rs1	2	SP ← Rs1
RTS	F	0	0	3	PC ← PMEMD(SP); SP ← SP+1
HALT	F	0	0	4	suspende seqüência de ciclos de busca e execução
NOP	F	0	0	5	nenhuma ação

4.1 Classes Funcionais de Instruções

A partir da Tabela 1, propõe-se a seguinte divisão das instruções em classes funcionais:

- As **instruções aritméticas** são **ADD, SUB, ADDI, SUBI, COMP;**
- As **instruções lógicas** são **AND, OR, XOR** e **NOT;**
- As **instruções de manipulação de bits** são **SL, SR;**
- As **instruções de movimentação de dados** são **LDLI, LDHI, LDIM, LD, ST, MOV, MOVSP, PUSH** e **POP;**
- As **instruções de controle de fluxo de execução** são **JPRGR, JSRGR, JPRG, JSRG, JPMI, JSRMI, RTS** e **HALT;**
- Existem três **instruções miscelâneas** , **STMSK, LDSP** e **NOP.**

4.2 Observações sobre a Semântica de Instruções no Processador R12

Algumas observações gerais e particulares sobre o conjunto de instruções são apresentadas a seguir.

- A arquitetura foi elaborada para privilegiar a simplicidade do conjunto de instruções, sem contudo sacrificar sua flexibilidade. Devido à grande limitação de todas as instruções possuírem exatamente o mesmo tamanho, instruções em geral disponíveis em processadores mais poderosos estão ausentes na R12. Contudo, foi tomado cuidado no projeto desta para que tal funcionalidade possa ser suprida de forma simples via o conceito de pseudo-instruções. **Pseudo-instruções** são instruções inexistentes em uma arquitetura, mas disponibilizadas ao programador em linguagem de montagem. Sua implementação mediante uso de uma seqüência de instruções existentes é feita no código objeto pelo programa montador. Por exemplo, uma instrução capaz de carregar uma constante imediata de 16 bits em um registrador não existe, mas pode ser implementada por uma seqüência de duas instruções, **LDLI** e **LDHI**. Seria possível a um programa montador para o processador R12 disponibilizar uma pseudo-instrução LDI com um operando imediato de 16 bits, gerando como código para esta uma seqüência de **LDLI** e **LDHI**.
- A instrução **COMP** opera interpretando números como valores em complemento de 2 (ou seja, números com sinal) pode ser usada em conjunto com os qualificadores de estado e as instruções de controle de fluxo para implementar comparações e laços baseados em testes de igualdade ou magnitude (menor ou maior). **Tarefa 1**: Proponha uma seqüência de instruções do processador R12 para implementar as seguintes pseudo-instruções: **JGTR**, **JGER**, **JLTR** e **JLER** (respectivamente, **salta se maior**, **se maior ou igual, se menor** e **se menor ou igual relativo**). Ver a Seção 4.3 para mais detalhes.

4.3 Funcionamento das Instruções de Controle de Fluxo

As instruções de controle de fluxo são os saltos, chamadas de subrotina, o retorno de subrotina (**RTS**) e a instrução de parada do processador (**HALT**). Instruções de salto e de chamada de subrotina (com prefixo **JP** e **JSR**, respectivamente) existem em três modos de endereçamento: relativo a registrador (aquelas cujo mnemônico possui sufixo **RGR**), a registrador (aquelas cujo mnemônico possui sufixo **RG**), e relativo com constante imediata de 12 bits (aquelas cujo mnemônico possui sufixo **MI**).

Como já mencionado ao apresentar a tabela de instruções da arquitetura R12, todos os saltos e chamadas de subrotina são condicionais. Um salto/chamada de subrotina somente é executado se a função Booleana denominada **JUMP** avaliar como verdadeira (valor lógico 1). **JUMP** é uma função Booleana de 12 variáveis. Quatro destas variáveis são os qualificadores ou *flags* de estado, **N**, **Z**, **C** e **V**. As variáveis restantes têm seus valores associados aos valores dos bits de um registrador interno do R12, denominado de **MASK**. Trata-se de um registrador de 8 bits que associa cada par de bits consecutivos a um dos *flags* de estado. Os bits de **MASK** denominam-se **M7**, **M6**, **M5**, **M4**, **M3**, **M2**, **M1** e **M0**. Os dois primeiros estão associados ao *flag* **N**, os dois seguintes ao *flag* **Z**, **M3** e **M2** estão associados ao *flag* **C** e os dois últimos ao *flag* **V**. Dada esta associação, a função **JUMP** é dada pela equação Booleana abaixo, onde operadores **.** e **+** correspondem às funções Booleanas **E lógico** e **OU lógico**, respectivamente.

$$JUMP = N.M7 + \overline{N}.M6 + Z.M5 + \overline{Z}.M4 + C.M3 + \overline{C}.M2 + V.M1 + \overline{V}.M0$$

O conteúdo do registrador **MASK** é definido pela instrução **STMSK**. Assim, a cada execução de um salto, o valor atual dos *flags* de estado e a última execução da instrução **STMSK** definem todas as condições para avaliar a função **JUMP**. Note-se que a partir da expressão soma de produtos Booleanos que definem o comportamento da função **JUMP**, se dois bits do registrador **MASK** associados a um mesmo *flag* estiverem em 1, isto implica na condição de salto verdadeira, independente do valor de qualquer dos *flags*. Esta é a forma de realizar saltos incondicionais. Por outro lado, se dois bits do registrador **MASK** associados a um mesmo *flag* estiverem em 0, isto implica que o valor do *flag* em questão é irrelevante para determinar a condição de salto.

A estrutura de especificação de desvios do processador R12 implica que cada instrução de desvio deve ser precedida por uma instrução de estabelecimento da condição de salto a usar. Um exemplo de salto incondicional em linguagem de montagem da arquitetura R12 é:

```
        STMSK   #0FFH   ; o valor imediato poderia ser C0H, 07H, F3H, mas não 60H ou 05H. Porquê?
        JPMI    XUXU    ; salto incondicional para o rótulo XUXU do programa
```

Além de saltos incondicionais, este esquema permite uma grande flexibilidade na especificação da condição a ser analisada para determinar se um salto será ou não tomado. A melhor maneira de visualizar as propriedades da técnica é através de exemplos. Seguem-se três casos comentados, a serem analisados pelo leitor.

```
Ex1JP:  STMSK   #10H    ; o valor imediato define a condição de salto, Z=0. Porquê?
        JPRG    R14     ; salta para o endereço contido no registrador R14, se última operação que afetou o
                        ; flag Z escreveu 0 nele
Ex2JSR: STMSK   #0AH    ; o valor imediato define a condição de salto, C=1 ou V=1. Porquê?
        JSRMI   XAXA    ; salto para a subrotina iniciando no rótulo XAXA se última operação aritmética gerou
                        ; vai-um (carry) ou transbordo aritmético (overflow)
```

```
Ex3JSR:  COMP     R3, R5     ; compara os valores de R3 e R5, realizando uma subtração e setando os flags.
         STMSK    #60H       ; o valor imediato define a condição de salto, maior ou igual. Porquê?
         JSRGR    R1         ; salta para o endereço contido no registrador R1 se R3 igual ou maior R5, em
                             ; complemento de 2.
```

5 REGISTRADORES DO PROCESSADOR R12 - BLOCO DE DADOS

O processador R12 conta com o seguinte conjunto de registradores:

- **IR** (*instruction register*): armazena o código de operação (*opcode*) da instrução atual e o(s) código(s) do(s) operando(s) desta. Possui 16 bits.
- **PC** (*program counter*): é o contador de programa. Possui 16 bits.
- **SP** (*stack pointer*): é o apontador de pilha. armazena o endereço do topo da pilha, controla a chamada e retorno de subrotinas. Deve ser inicializado a cada programa que o empregue com a instrução **LDSP** (carrega endereço do topo da pilha). Possui 16 bits.
- Um banco de registradores contendo 16 registradores de uso geral, cada um de 16 bits., denominados **R0** a **R15**. O banco de registradores tem uma porta de escrita e duas de leitura. Isto significa que é possível escrever em apenas um registrador por vez, porém é possível fazer, em paralelo com a escrita mencionada, duas leituras simultâneas, colocando o conteúdo de um registrador no barramento de saída fonte 1 e o conteúdo de outro registrador (ou o mesmo) no barramento de saída fonte 2.
- **MASK**: é o registrador de máscara para a condição de saltos, já discutido na Seção 4.3. Possui 8 bits.
- ***Flags*** **N**, **Z**, **C** e **V**: registradores de 1 bit já discutidos anteriormente.
- **MDR** (*memory data register*): é registrador que recebe dados provenientes da memória ou de outro local que produza um valor que necessite ser escrito no banco de registradores no último ciclo de uma instrução. Possui 16 bits.

Normalmente, sempre existe a necessidade de existirem registradores adicionais, dependendo da organização implementada. Na organização proposta aqui, o valor do registrador PC é atualizado após a busca, mas o novo valor não é imediatamente escrito em **PC**, mas em um registrador temporário **NPC**, denominado *new program counter*. Apenas no último ciclo de relógio de uma instrução o valor de **NPC** (ou outro, dependendo da instrução) é escrito no **PC**. Por razões similares, o mesmo ocorre com o apontador de pilha e então associa-se ao registrador **SP** um outro, **NSP**, denominado de *new stack pointer*. Os valores lidos do banco de registradores são armazenados nos registradores **RA** e **RB**, e o valor obtido pela execução de uma dada operação lógico-aritmética é armazenado no registrador **RALU**. Mais detalhes são apresentados na próxima Seção.

6 ORGANIZAÇÃO DO BLOCO DE DADOS DO PROCESSADOR R12

A execução de cada instrução neste processador requer 2 a 5 ciclos de relógio. Cada ciclo executa um conjunto limitado de partes de uma instrução e são assim denominados:

- **Ciclo 1: busca da instrução.** Comum a todas as instruções.
- **Ciclo 2: decodificação e leitura de registradores.** Comum a todas as instruções.
- **Ciclo 3: operação com a ALU.** Comum a quase todas as instruções.
- **Ciclo 4: acesso à memória**. Realizado conforme a instrução
- **Ciclo 5: atualização do banco de registradores ("write-back")**. Realizado conforme a instrução.

Esta Seção discute uma proposta de organização para o bloco de dados do processador R12. A Figura 2 mostra uma proposta completa de bloco de dados para o processador R12.

O bloco de dados necessita **12** sinais de controle, organizados em **3** classes:

- habilitação de escrita em registradores (9): ***wpc***, ***wsp***, ***wreg***, ***wnz***, ***wcv***, ***one***, ***two***, ***three***, ***four***.
- controle de leitura/escrita na memória externa (2): ***ce*** e ***rw***.
- a operação que a unidade lógica-aritmética executa e o controle dos multiplexadores, resultante da decodificação das instruções (1): ***i***

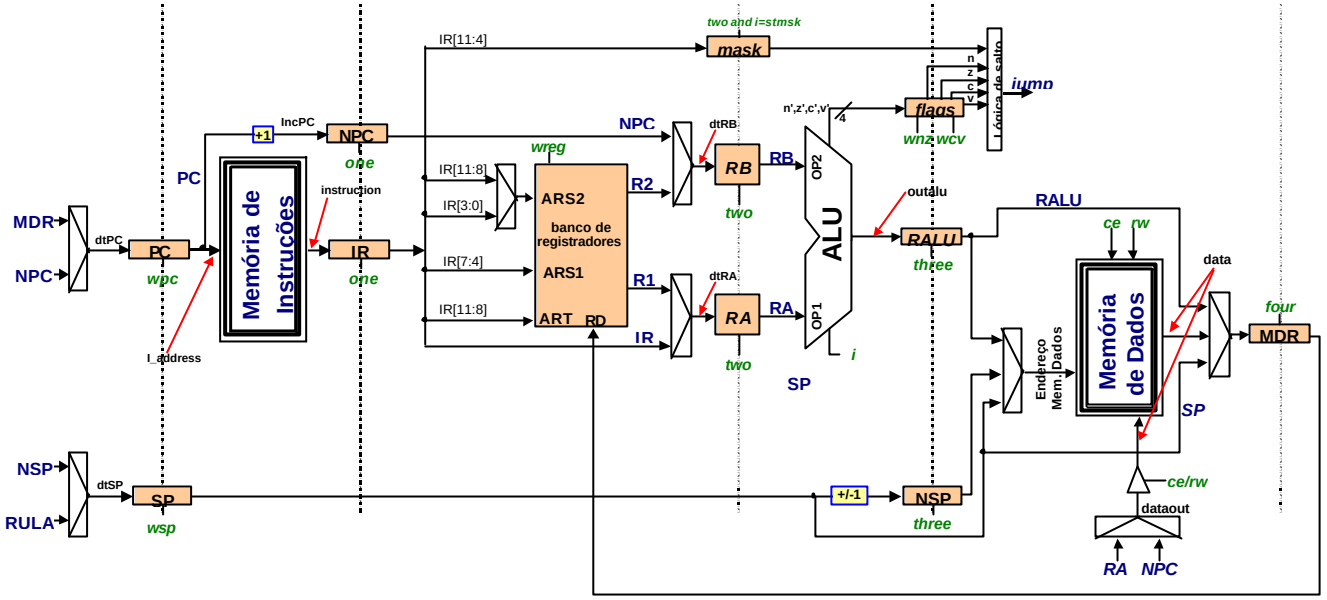


Figura 2 - Bloco de dados completo, com as memórias externas (de instruções e de dados) mostradas para fins de clareza. Estão representados todos os 12 sinais que o bloco de controle deve gerenciar (em verde, itálico). Os sinais de clock e reset não estão representados, porém são utilizados por todos os registradores.

Para se ter uma idéia geral em mais detalhe da implementação do bloco de dados do processador R12, restaria apresentar a organização interna do banco de registradores e da unidade aritmética e lógica. A Figura 3 ilustra a organização do banco de registradores, sob forma de um diagrama de blocos. A ALU será discutida em parte na Seção 8.1.

A organização do banco de registradores inclui os 16 registradores em si e a implementação das duas portas de leitura e da porta de escrita, bem como da decodificação do endereço de escrita para geração da habilitação de escrita do registrador em causa. As portas de leitura consistem de multiplexadores (16X16):(1:16), controladas pelos endereços de leitura.

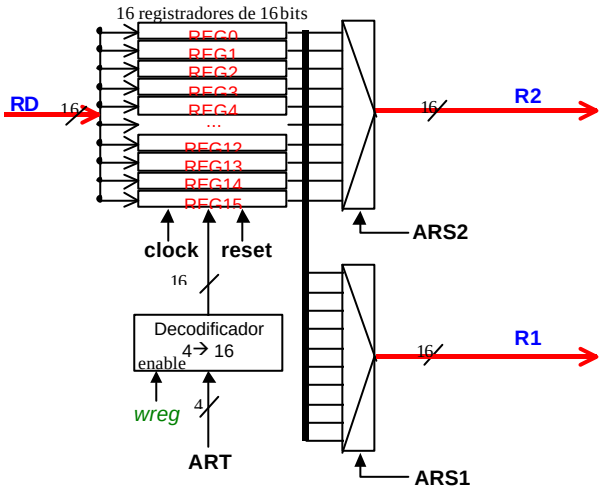


Figura 3 - Diagrama em blocos do banco de registradores de uso geral.

Os sinais de controle dos multiplexadores dependem da instrução corrente. Recomenda-se localizar os controles dos multiplexadores na Figura 2. Os sinais de controle são precedidos do sufixo '*uins.*", que designa a palavra de microinstrução. Abaixo se mostra um exemplo de codificação VHDL possível de ser usada para implementar o multiplexador para geração do endereço da memória de dados.

```
d_address  <=  nsp   when uins.i=rts or uins.i=pop else
               sp    when inst_subrt='1' or uins.i=push else
               ralu;


-- onde:
inst_subrt <= '1' when uins.i=jsrgr or uins.i=jsrg or uins.i=jsrmi else '0';
```

6.1 Instruções Aritméticas e Lógicas a Registrador, de Manipulação de Bits, MOV e MOVSP

Discute-se aqui o subconjunto de estruturas do Bloco de Dados R12 usado para permitir a implementação de instruções aritméticas e lógicas com endereçamento a registrador (**ADD, SUB, AND, OR, XOR, COMP, NOT, MOV, MOVSP**) e as de manipulação de bits (**SL, SR**). Nos 5 ciclos de execução destas instruções, têm-se as seguintes ações, que devem ser interpretadas referenciando-se à Tabela 1 e à Figura 2:

Tabela 2 – Funcionamento das instruções lógicas e aritméticas a registrador, de manipulação de bits MOV e MOVSP.

CICLO	AÇÃO
1	- Busca a instrução e altera **NPC: IR ← PMEM(PC); NPC ← PC + 1**.
2	- Armazena em **RA** e **RB** o conteúdo dos registradores fonte **RS1** e **RS2** respectivamente. Mesmo nas instruções unárias, carrega-se ambos registradores, e depois a ALU opera sobre apenas um. A operação de comparação carrega o registrador **RS2** a partir do endereço contido em **IR [11:8]**.
3	- Realiza a operação especificada entre os registradores, grava o resultado em **RALU** e atualiza flags relevantes. A instrução **MOVSP** não usa a ALU.
4	- O valor a ser escrito no banco de registradores, se existir, é gravado no registrador **MDR**.
5	- Atualiza o valor do contador de programa (**PC**) e escreve no banco de registradores o resultado da operação, exceto no caso da instrução **COMP**.

- A instrução de comparação aritmética (**COMP**) executa uma subtração em complemento de 2 e despreza o resultado da operação (ou seja, não o armazenando em qualquer lugar), restando como efeito de sua execução apenas a escrita de valores nos qualificadores, refletindo condições de igualdade ou de maior ou menor que.
- As instruções de deslocamento de bits usam **RT** e deslocam os bits deste registrador de uma posição usando a ALU. Os bits de um dos extremos da palavra são perdidos (em **SL**, o bit mais à esquerda, em **SR** o bit mais à direita). Por exemplo, supor que o registrador de uso geral **R3** contém o valor binário **1111000000000000**. O resultado de executar **SL R4, R3** é colocar **1110000000000000** em **R4**.

6.2 Instruções com Operando Imediato

A instruções com modo de endereçamento imediato (**ADDI, SUBI, LDLI, LDHI**) implicam em um dado registrador destino (**RT**) receber o resultado de uma operação entre este registrador e uma constante de 8 bits. As operações em modo de endereçamento imediato são:

- soma/subtração em modo imediato (**ADDI, SUBI**): soma/subtração do conteúdo de um dado registrador a uma constante de 8 bits: **Rt ← Rt +/- constante**. **Importante**: a execução da instrução implica completar com zeros os 8 bits mais significativos da constante, gerando assim o necessário valor de 16 bits.
- carga da parte baixa/alta de um registrador (**LDLI, LDHI**): **Rt ← RtH & constante** ou **Rt ← constante & RtL** (o registrador destino recebe a constante na parte baixa/alta e mantém a parte alta/baixa inalterada). **Importante**: para inicializar um registrador com uma constante de 16 bits deve-se utilizar uma seqüência de 2 instruções, **LDLI** e **LDHI**. Para ler/escrever um dado contido em um endereço qualquer de memória é necessário empregar 3 instruções em linguagem de montagem: as duas primeiras carregam em um registrador a parte baixa e alta de um endereço (**LDLI** e **LDHI**, respectivamente) e a terceira instrução realiza a leitura/escrita (**LDIM, LD** ou **ST**). Isto ocorre porque não há instruções que possam conter um endereço completo de memória (16 bits). Exemplo:

```
XOR     R0, R0, R0 ; zera o conteúdo do registrador R0
LDHI    R1, #03H
LDLI    R1, #27H  ; armazena no registrador R1 o valor 0327H
LD      R5, R0, R1 ; armazena em R5 o conteúdo do endereço 0327H de memória
```

As instruções **ADDI, SUBI, LDLI, LDHI** são executadas de forma semelhante às descritas na Seção anterior. As seguintes ações são realizadas:

Tabela 3 – Funcionamento das instruções com operando imediato.

CICLO	AÇÃO
1	- Busca a instrução e altera **NPC: IR ← PMEM(PC); NPC ← PC + 1**.
2	- Armazena em **RB** o conteúdo do registrador que vai ser operado e modificado (**RT**). - Armazena em **RA** o conteúdo do **IR** (mesmo usando-se apenas parte de **IR** armazena-se todo ele).
3	- Realiza a operação especificada pela instrução, grava **RALU** e *flags*.
4	- O valor a ser escrito no banco de registradores, se existir, é gravado no registrador **MDR**.
5	- Atualiza o valor do contador de programa (**PC**) e escreve no banco de registradores o resultado da operação.

6.3 Instruções de Leitura da Memória de Dados

As instruções de leitura da memória de dados são executadas em 5 ciclos de relógio incluindo as ações:

Tabela 4 – Funcionamento das instruções de leitura da memória.

CICLO	AÇÃO
1	- Busca a instrução e altera **NPC: IR ← PMEM(PC); NPC ← PC + 1**.
2	- Armazena em **RB** o conteúdo do registrador especificado por **Rs2**. - Armazena em **RA** o conteúdo do registrador especificado por **Rs1** no caso de **LD** e **IR** no caso de **LDIM**.
3	- Determina o endereço do dado a ser lido da memória, grava em **RALU.**
4	- Lê o dado da memória, endereçado por **RALU**, e grava no registrador **MDR.**
5	- Atualiza o valor do contador de programa (**PC**) e escreve no banco de registradores o resultado o conteúdo do registrador **MDR**.

6.4 Instruções de Escrita na Memória de Dados

A instrução de escrita em memória de dados é executada em 5 ciclos, incluindo as seguintes ações:

Tabela 5 – Funcionamento da instrução ST.

CICLO	AÇÃO
1	- Busca a instrução e altera **NPC: IR ← PMEM(PC); NPC ← PC + 1**.
2	- Armazena em **RA** o dado a ser gravado em memória; - Armazena em **RB** o endereço do dado a escrever na memória (**RT**).
3	- Apenas transfere o conteúdo de **RB** para **RALU.**
4	- Armazena na posição de memória apontada por **RALU** o conteúdo do registrador **RA.**
5	- Atualiza o valor do contador de programa (**PC**).

6.5 Instruções de Salto Condicional

As instruções **JPRGR, JPRG, JPMI** são executadas todas em 5 ou em apenas 3 ciclos de relógio, dependendo do valor do sinal **JUMP**. Caos este esteja com valor '0' a instrução de salto condicional termina em 3 ciclos, caso contrário a instrução é executada em 5 ciclos.

Nos até 5 ciclos necessários para executar estas instruções, tem-se as seguintes ações:

Tabela 6 – Funcionamento das instruções de salto condicional.

CICLO	AÇÃO
1	- Busca a instrução e altera **NPC: IR ← PMEM(PC); NPC ← PC + 1**.
2	- Armazena em **RB** o registrador **NPC**, em **RA** o operando (**Rs1** ou **IR**).
3	- Se **JUMP=0**, atualiza o valor do contador de programa (**PC**) com **NPC** e **FIM da INSTRUÇÃO**; - Senão, calcula o endereço de salto, armazenando-o em **RALU**.
4	- Escreve em **MDR** o endereço da instrução a ser executada após o salto, atualmente em **RALU**.
5	- Atualiza o valor do contador de programa (**PC**) com o conteúdo de **MDR**.

6.6 Instruções de Chamada de Subrotina e PUSH

Do ponto de vista prático, estas são as instruções mais complexas quanto ao uso de recursos específicos do processador R12. Por esta razão, algumas arquiteturas de processadores comerciais optam por não disponibilizar estes recursos ao nível de programação em linguagem de montagem, devendo todo o processamento de implementação e controle de pilha ser realizado explicitamente em software. Um exemplo clássico é a arquitetura MIPS, usada por muito tempo como Unidade Central de Processamento nas estações de trabalho da empresa Silicon Graphics.

Na R12, todas as instruções que usam a pilha o fazem através do registrador apontador de pilha **SP**. As únicas que não usam pilha mas usam o **SP** são as instruções **LDSP** e **MOVSP** que inicializa este registrador e que o copia para um registrador de uso geral, respectivamente. Toda instrução que usa a pilha faz acesso à memória para leitura ou escrita.

Uma pilha é uma estrutura de dados onde o acesso sempre se dá no mesmo ponto, seja para escrita, seja para leitura. Todas as operações em uma pilha se dão sobre a posição denominada *topo da pilha.* Em processadores que usam memória com acesso aleatório (RAM), esta estrutura de dados deve ser simulada. A forma comum de fazer esta simulação em hardware é definir um registrador especial, denominado apontador de

pilha (em inglês *stack pointer*) para conter o endereço da memória de acesso aleatório onde se encontra o topo da pilha. O registrador **SP** é automaticamente atualizado a cada operação de inserção e/ou remoção de um dado da pilha. A atualização do apontador de pilha normalmente usa decrementos sucessivos para tarefas de empilhamento e incrementos sucessivos para tarefas de desempilhamento, para prover a visão intuitiva de que uma pilha de objetos cresce para cima (endereços menores) e decresce para baixo (endereços maiores). Para tanto, duas estratégias equivalentes são possíveis para implementar o processo de atualização do apontador. A primeira é usar pré-decremento seguido de empilhamento e desempilhamento seguido de pós-incremento, em cujo caso o apontador de pilha possui sempre o endereço do elemento do topo (esta é a estratégia adotada na R12, por ser mais intuitiva). A segunda estratégia é usar empilhamento seguido de pós-decremento e pré-incremento seguido de desempilhamento, em cujo caso o apontador de pilha está sempre apontando a primeira posição livre da pilha, imediatamente acima do topo. A Figura 4 ilustra o funcionamento da pilha, conforme previsto na arquitetura R12.

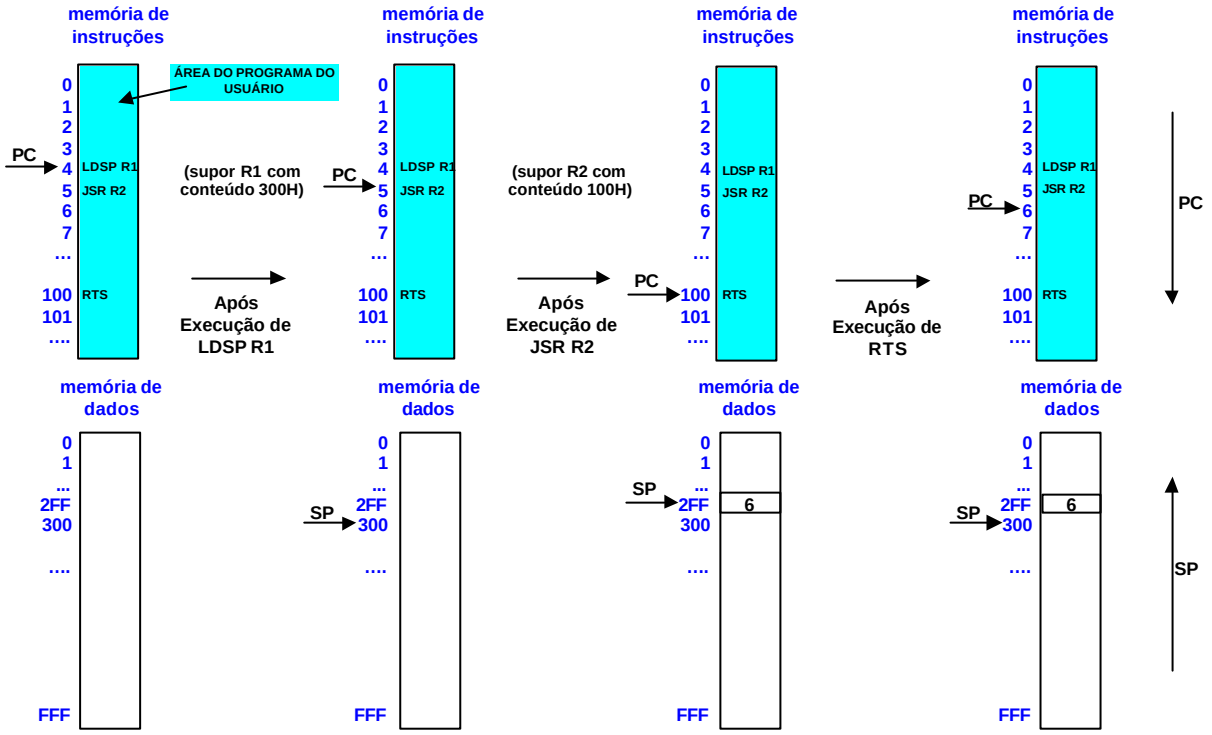


Figura 4 - Ilustração do funcionamento da pilha na arquitetura R12, usando o registrador de controle SP. O exemplo mostra a execução de instruções de chamada e retorno de subrotina. O programa é armazenado na memória de instruções. A pilha cresce no sentido inverso ao do avanço do registrador PC, e está situada na memória de dados. O conteúdo do registrador SP sempre aponta para o elemento atualmente no topo da pilha. Note-se o gerenciamento automático do registrador SP.

As instruções **JSRGR, JSRG, JSRMI** são executadas em 3 ou 5 ciclos de relógio. Como no caso das instruções de salto, caso o sinal **JUMP** esteja com valor '0' a instrução de chamada condicional a subrotina termina após três ciclos de relógio. Nos até 5 ciclos necessários para executar estas instruções, tem-se as seguintes ações:

Tabela 7 – Funcionamento das instruções de chamada de subrotina e da instrução PUSH.

CICLO	AÇÃO
1	- Busca a instrução e altera **NPC: IR ← PMEM(PC); NPC ← PC + 1**.
2	- Armazena em **RB** o registrador **NPC**, em **RA** o operando (**Rs1** ou **IR**).
3	- Se **JUMP=0**, atualiza o valor do contador de programa (**PC**) com **NPC** e **FIM da INSTRUÇÃO**; - Senão, calcula o endereço de salto, armazenando-o em **RALU**. Além disto, decrementa o **SP** e coloca o resultado em **NSP**.
4	- Armazena na posição de memória apontada por **NSP** o **NPC** ou o **RA** (caso instrução **PUSH**). Armazena em **MDR** o endereço de salto (*se diferente de* **PUSH**).
5	- Atualiza o **PC** conforme o endereço armazenado em **MDR** (*se diferente de* **PUSH**) ou com **NPC**. Atualiza o **SP** com o valor de **NSP**.

6.7 Demais Instruções

É relativamente fácil, a partir da discussão anterior, inferir o funcionamento das instruções restantes. A instrução **STMSK** é executada em 3 ciclos: busca de instrução, gravação de valor imediato de 8 bits no registrador

MASK e atualização do **PC**. A instrução **LDSP** é executada em 5 ciclos: busca de instrução, leitura do banco de registradores, gravação do valor a carregar no **SP** em **RALU**, ciclo morto para acesso à memória e atualização registrador **SP** e do **PC**. As instruções **POP** e **RTS** são semelhantes às de chamada de subrotina e **PUSH**, com a diferença que nelas se faz incremento do **SP** e não decremento, o incremento é feito após o acesso à memória e não antes e são executadas em 5 ciclos. A instrução **HALT** é executada em três ciclos, sendo que no segundo, a decodificação do **HALT** faz com que a máquina para de executar instruções e sinalize isto como uma condição de exceção. A instrução **NOP** é executada em 3 ciclos de relógio, busca, decodificação e atualização do **PC** com **NPC**.

6.8 Resumo das Ações Executadas pela ALU

A Tabela 8 ilustra a estrutura de hardware para efetuar operações da ALU. Esta última é um hardware combinacional que possui duas entradas de dados e uma saída de dados. Além disto, a ALU possui uma entrada de controle para informar a operação a ser realizada em cada instante (entrada com nome **I**), e quatro saídas de 1 bit, representando os qualificadores gerados a cada operação lógica ou aritmética. Estas saídas são entradas para registradores de 1 bit que guardam estes qualificadores, dependendo de sinais de controle provenientes do bloco de controle, que escolhe o momento adequado para armazenar estes valores. No processador R12, a exemplo do que ocorre no processador Cleópatra, há necessidade de apenas dois sinais de controle para controlar a carga dos quatro qualificadores pois ou todos os qualificadores são escritos ou apenas os dois qualificadores **N** e **Z** são escritos, devido à semântica explicitada na Tabela 1.

O ciclo de operação com a ALU é comum a quase todas as operações. O resultado da operação com a ALU é armazenado no registrador **RALU**. Os qualificadores **C** e **V** são gerados com valor coerente a cada operação aritmética apenas. A Tabela 8 define quais devem ser as entradas da ALU, **Op1** e **Op2**, respectivamente (definidas na Figura 2), conforme a instrução, no início do terceiro ciclo de cada instrução.

Tabela 8 – Entradas da ALU no início do terceiro ciclo de operação, para cada instrução.

Instruções	RA	RB
ADD, SUB, AND, OR, XOR, LD, COMP	R1	R2
ADDI, SUBI, LDLI, LDHI, LDIM	IR	R2
ST	R1 (não usado)[2]	R2
NOT, SL, SR, MOV, JPRG, JSRG	R1	-
JPRGR, JSRGR	R1	NPC
JPMI, JSRMI	IR	NPC
LDSP, PUSH	R1 (não usado)[3]	-
MOVSP, STMSK, POP, RTS	-	-
NOP, HALT	Não se aplica	

7 NÚMERO DE CICLOS PARA AS INSTRUÇÕES DA ORGANIZAÇÃO R12

Dada a descrição da organização do bloco da dados da R12 apresentada na Seção anterior, é possível sumarizar o tempo de execução de todas as instruções em termos de ciclos de relógio tomados, o que é então mostrado na Tabela 9 abaixo.

Tabela 9 – Número de ciclos gastos para buscar e executar instruções na organização do processador R12.

INSTRUÇÃO	NUMERO DE CICLOS	INSTRUÇÃO	NUMERO DE CICLOS
ADD	5	MOV	5
SUB	5	MOVSP	5
AND	5	STMSK	3
OR	5	JPRGR	3(JUMP=0)/5(JUMP=1)
XOR	5	JSRGR	3(JUMP=0)/5(JUMP=1)
ADDI	5	JPRG	3(JUMP=0)/5(JUMP=1)
SUBI	5	JSRG	3(JUMP=0)/5(JUMP=1)
LDLI	5	JPMI	3(JUMP=0)/5(JUMP=1)
LDHI	5	JSRMI	3(JUMP=0)/5(JUMP=1)

[2] Embora **R1** não seja usado pela ALU nesta operação, o valor tem de estar disponível em **RA** para garantir que no ciclo seguinte pode-se usar ele para escrita na memória.

[3] Embora **R1** não seja usado pela ALU nesta operação, o valor tem de estar disponível em **RA** para garantir que no ciclo seguinte pode-se usar ele para escrita na memória ou no **SP**.

LDIM	5	POP	5
LD	5	PUSH	5
ST	5	LDSP	5
COMP	5	RTS	5
NOT	5	HALT	3
SL	5	NOP	3
SR	5		

8 BLOCO DE CONTROLE

Para comandar a execução de instruções neste processador, define-se uma máquina de estados de controle. A Figura 5 ilustra esta máquina de estados em linhas gerais, onde o próximo estado é função apenas do estado atual e da instrução armazenada no registrador **IR**. Também se indica nesta Figura quais registradores são alterados em cada estado. Não se mostra as ativações dos sinais de controle dos multiplexadores, ou as operações da ALU.

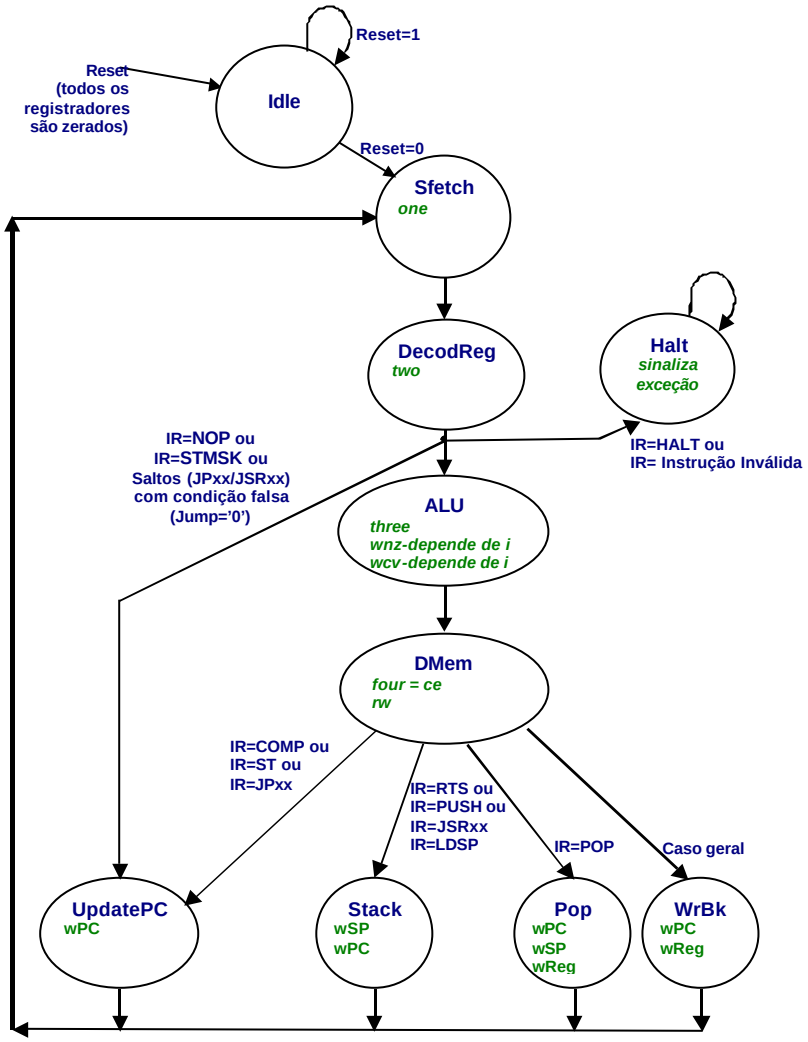


Figura 5 - Máquina de estados de controle para organização R12 proposta.

A função dos 10 estados mostrados na Figura é:

- **Idle:** estado inicial após o reset, serve para garantir que a primeira borda de subida de relógio após este sinal defina o início da operação do processador R12;
- **Fetch:** primeiro ciclo, busca de instrução;
- **DecodReg:** segundo ciclo, leitura dos registradores fonte e teste do sinal *jump*;
- **Halt:** terceiro ciclo, sinaliza exceção e aguarda reset;
- **ALU:** terceiro ciclo, operação com a ALU;
- **DMem:** quarto ciclo, acesso à memória de dados;

- **UpdatePC:** terceiro ciclo ou quinto ciclo, apenas atualiza valor do **PC**;
- **Stack:** quinto ciclo, operações com pilha exceto POP;
- **Pop:** quinto ciclo, retirada de dados da pilha;
- **WrBk:** quinto ciclo para a maioria da instruções, onde se escreve o resultado no banco de registradores e atualiza-se o contador de programa;

8.1 Decodificação das instruções / Definição da operação da ALU

A primeira ação a ser realizada no bloco de controle após a busca de instrução é a decodificação desta. Como sugestão, pode-se definir em um *package* da descrição do processador um tipo enumerado contendo todas as possíveis instruções do processador R12. O código VHDL associado seria o seguinte:

```
type instrucao is
    ( add, sub, i_and, i_or, i_xor, addi, subi, ldli, ldhi, ldim, ld, st,
      comp, i_not, sl, sr, mov, movsp, stmsk, jprgr, jsrgr, jprg, jsrg, jpmi,
      jsrmi, pop, push, ldsp, rts, halt, nop,
      invalid_instruction );
```

Note-se que existe um código especial para representar uma instrução inválida que pode aparecer, por exemplo, caso o programa tente executar um código ao ler de uma área de dados, devido a erros de programação.

- **Decodificação das instruções**: Um exemplo de trecho de código VHDL referente à decodificação de instruções aparece abaixo. Notar o uso do código **`invalid_instruction`**. Isto será útil para gerar a exceção associada mencionada no início deste documento.

```
i <= add   when ir(15 downto 12)=x"1"  else
     sub   when ir(15 downto 12)=x"2"  else
     ...
     st    when ir(15 downto 12)=x"B" else
     comp  when ir(15 downto 12)=x"B" and ir(3 downto 0)=x"0" else
     ...
     halt  when ir(15 downto 12)=x"F" and ir(3 downto 0)=x"4" else
     nop   when ir(15 downto 12)=x"F" and ir(3 downto 0)=x"5" else
     invalid_instruction;

uins.i <= computa_cod_alu(i);       --- uma função determina a operacao da ALU a partir de i.
```

9 TRABALHO PRÁTICO A SER DESENVOLVIDO

Implementar em VHDL o processador multi-ciclo, descrito nas Seções anteriores. O bloco de dados deve ter uma descrição semelhante ao processador Cleópatra, entretanto o bloco de controle deve ser implementado conforme esboçado na Seção 8, através de uma máquina de estados. A nota final deste trabalho dará **ENORME ÊNFASE** à execução correta da simulação. Assim, aconselha-se testar cada módulo implementado do hardware. A nota de uma descrição completa sem nenhuma simulação tenderá a 0 (zero), enquanto que a nota de uma descrição incompleta com boas simulações de cada módulo implementado tenderá ao máximo valor de nota. As regras do jogo são:

- O trabalho de implementação pode ser realizado por até 3 alunos (*grupo*). Mais do que 3 alunos no grupo implicará automaticamente na não avaliação do trabalho, e conseqüente nota.
- A apresentação será oral, teórico-prática, frente ao computador, onde o *grupo* deverá explicar ao professor o projeto, a simulação e a implementação. A avaliação de cada membro do grupo será individual, baseada no desempenho durante a apresentação. Questões individuais serão colocadas aos membros do grupo. Após a apresentação, entregar ao professor um disquete com o projeto (fonte do processador, fonte do *test_bench* e programas de teste em código objeto e linguagem de montagem, ambos **adequadamente comentados**).
- Cada *grupo* deve desenvolver uma aplicação (um programa contendo, no mínimo, 40 instruções em linguagem de montagem) para a R12, com utilização de pelo menos duas subrotinas e pelo menos uma chamada aninhada de subrotina. **Esta aplicação deve ser validada via o montador/simulador da R12, e deve ser entregue no prazo máximo de 15 dias após a disponibilização do ambiente de desenvolvimento de software R12. Consulte a homepage da disciplina para saber o prazo máximo para a sua Turma específica. Esta aplicação valerá 25% da nota do Trabalho Prático**. A avaliação da aplicação considerará a criatividade na escolha do problema a resolver, o estilo de programação, a capacidade de acrescentar comentários úteis no código fonte, a extensão do programa, bem como a otimalidade do código.
- As apresentações ocorrerão em data divulgada na homepage da disciplina (2/3 aulas para cada turma). Sistemática: metade dos grupos no primeiro dia, metade no segundo. Para marcar dia contatar o professor, desde que o projeto esteja avançado (tipicamente, 50% pronto). A este caberá julgar se o trabalho está

adiantado o suficiente para permitir a marcação da data de apresentação. As demais apresentações serão marcadas pelo professor no máximo 7 dias antes da primeira apresentação, via sorteio entre os grupos restantes. As apresentações ocorrerão nos dias indicados na homepage (3 aulas). Sistemática: Cada dia um terço dos grupos apresentará o seu trabalho.

- Todos os trabalhos devem ser entregues para o professor no primeiro dia de avaliação em meio magnético (ver dia específico para cada turma na *homepage*), conforme especificado anteriormente, durante o período da aula. Um representante do grupo deverá assinar a ata de entrega do trabalho. Trabalhos entregues fora desta data e do horário estipulado não serão avaliados. Não serão permitidas substituições, modificações ou alterações futuras no trabalho após a entrega para o professor.
- Composição média da nota do Trabalho:

BLOCO DE DADOS	BLOCO DE CONTROLE	Estrutura Geral e test_bench	Simulação das instruções básicas	Simulação de outras instruções	Aplicação R12
20%	20%	5%	15%	15%	25%

Recomenda-se desenvolver inicialmente o bloco de dados, iniciar o bloco de controle, realizando-se simulações parciais para verificar a implementação. De nada adianta um código dito completo, caso não se tenha realizado simulações corretas.

10 PROGRAMA PARA TESTAR TODAS AS INSTRUÇÕES DA R12

O código objeto mostrado na Figura 6 corresponde a um exemplo de arquivo texto que é lido pelo *test bench* durante a simulação do processador. Este arquivo contém *n* linhas, contendo cada uma 9 caracteres na forma "xxxx yyyy", onde xxxx é o endereço de memória de 16 bits (4 dígitos hexadecimais) e yyyy é a instrução ou dado de 16 bits (4 dígitos hexadecimais) a ser carregada neste endereço. O arquivo de teste é carregado na memória quando o reset é ativado, no início da simulação.

Recomenda-se **escrever os programas em linguagem de montagem** (*assembly*), gerando-se o código objeto automaticamente, a partir do montador/simulador. A ferramenta de simulação, assim como documentação de como utilizar a ferramenta encontra-se na página da disciplina.

A Figura 7 mostra a janela do simulador. A esquerda desta figura está apresentada a memória de instruções, contendo em cada linha a instrução em *assembly*, o endereço da posição da memória e o código objeto. Ao centro estão inseridas a memória de dados e a tabela de símbolos. Na primeira, aparecem dados definidos pelo usuário ou escritos pelo programa, e na segunda aparecem o nome dos símbolos, seu endereço de memória e o seu valor. A direita da figura estão localizados os registradores de uso geral e os registradores IR, PC e SP. Na parte inferior são ilustrados os botões de controle *Step*, *Run*, *Pause*, *Stop* e *Reset* e as opções de velocidade *Slow*, *Normal* e *Fast*, com as interpretações óbvias. Os qualificadores de estado encontram-se na parte inferior à direita.

A ferramenta de montagem tem como entrada o nome do programa em linguagem descrito em linguagem de montagem (<file>.asm) e o nome da arquitetura. São gerados três arquivos de saída, um com extensão **.hex** para download na placa de prototipação, um com extensão **.sym** – para uso do simulador e um com extensão **.txt** para uso no test_bench do simulador Active-HDL.

A ferramenta de montagem é transparente para o usuário, pois a mesma está integrada ao simulador. Os três arquivos de saída são gerados no momento da leitura de um arquivo fonte, como subproduto da execução do montador. Erros encontrados durante a execução de uma das fases do montador são salvos em um arquivo de mensagens. Este arquivo é lido pelo simulador após a execução do montador a fim de que os erros sejam apresentados ao usuário e não se prossiga a simulação. Erros na execução do montador não possibilitam a simulação, porque os mesmos indicam que as instruções da aplicação em linguagem de montagem não condizem com as instruções existentes na arquitetura. Ao término da simulação de um programa, algumas estatísticas são geradas sobre a execução, tais como número de clocks gastos pelo programa, total de instruções executadas e o CPI (Clocks Por Instrução) resultante.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; ARQUITETURA R12 - teste de todas as instruções
;; Autor: Ney Calazans (calazans@inf.pucrs.br)
;;
;; (exaustivo quanto ao total de instruções, não exaustivo quanto
;;  às condições de teste de cada instrução em si)
;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.code
                  LDLI     R1,#0ABH          ; flags NZCV=0000
                  LDHI     R1,#01H           ; endereço 01ABh para topo da pilha
                  LDSP     R1                ;
                  ;; teste de algumas instruções
                  LDLI     R1,#0F3H          ;
                  LDHI     R1,#23H           ; R1<=23F3h
                  LDLI     R8,#52H           ;
                  LDHI     R8,#0E2H          ; R8<=E252, N=1
                  LDLI     R15, #00H         ; R15<=0, Z=1
                  LDHI     R15, #8FH         ; R15<=8F00h, Z=0, N=1
                  ADD      R3,R1,R8          ; R3<=23F3h+E252h=0645h, N=0, C=1
                  SUB      R4,R1,R8          ; R4<=23F3h-E252h=41A1h, C=0
                  SUB      R5,R1,R1          ; R5<=0, Z=1, C=1
                  AND      R6,R8,R1          ; R6<=E252h and 23F3h=2252h
                  OR       R7,R1,R8          ; R7<=23F3h or E252h=E3F3h, N=1
                  ADDI     R1, #0ABH         ; R1<=23F3h+00ABh=249Eh, N=0
                  SUBI     R1, #0ABH         ; R1<=249Eh-00ABh=23F3h
                  STMSK    #0FFH             ; gera condição de salto incondicional
                  JSRMI    #somavet          ; salto p/ subrotina somavet SP<=01AAh
                  LDLI     R1,#end3          ;
                  LDHI     R1,#end3          ; carrega ponteiro para vetor resultado
                  LDIM     R2,R1, #3         ; carrega em R2 o quarto elemento do vetor 016AH
                  MOV      R3,R2             ; copia o valor de R2 em R3 (061AH)
                  NOP                        ; testa NOP
                  MOVSP    R4                ; testa MOVSP, R4 deve receber 01ABH
                  STMSK    #0FFH             ; salto incondicional para a subrotina testeshift
                  JSRMI    #testeshift       ;
                  HALT                       ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Um exemplo útil - soma de dois vetores
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
somavet:          PUSH     R1                ; salva regs alterados aqui
                  PUSH     R2                ;
                  PUSH     R3                ;
                  PUSH     R4                ;
                  PUSH     R5                ;
                  PUSH     R6                ;
                  PUSH     R7                ;
                  XOR      R5,R5,R5          ; R5 <- 0
                  LDLI     R1,#end1          ;
                  LDHI     R1,#end1          ;
                  LDLI     R2,#end2          ;
                  LDHI     R2,#end2          ;
                  LDLI     R3,#end3          ;
                  LDHI     R3,#end3          ; carrega os ponteiro para os três vetores
                  LDLI     R4,#n             ;
                  LDHI     R4,#n             ;
                  LD       R4,R4,R5          ; carrega o número de elementos
loop:             LD       R6,R1,R5          ;
                  LD       R7,R2,R5          ;
                  ADD      R7,R6,R7          ;
                  ST       R3,R7             ; *end3 <- *end1 + *end2
                  ADDI     R5,#01H           ; incrementa deslocamento - buscar *end1 e *end2
                  ADDI     R3,#01H           ; incrementa o endereço do vetor *end3
                  SUBI     R4,#01H           ;
                  STMSK    #10H              ; z' (n/n'/z/z'/c/c'/v/v')
                  JPMI     #loop             ;
                  POP      R7                ;
                  POP      R6                ;
                  POP      R5                ;
                  POP      R4                ;
                  POP      R3                ;
                  POP      R2                ;
                  POP      R1                ;
testeJSRGR:       RTS                        ; final da subrotina e início e final de outra
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; teste de shifts e outras instruções
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
testeshift:       PUSH     R1                ; empilha os registradores utilizados
                  PUSH     R2                ; na subrotina
                  PUSH     R3                ;
                  LDLI     R1,#55H           ;
                  LDHI     R1,#AAH           ; R1<= AA55h = 1010 1010 0101 0101
                  NOT      R1,R1             ; R1<= 0101 0101 1010 1010 = 55AAh
                  SL       R2,R1             ; R2<= 1010 1011 0101 0100 = AB54h
                  SL       R2,R2             ; R2<= 56A8h, bit mais à esq perdido
                  SR       R3,R1             ; R3<= 0010 1010 1101 0101 = 2AD5h
                  SR       R3,R3             ; R3<= 0001 0101 0110 1010 = 156Ah
                  COMP     R3,R2             ; NZCV=1000 indica (156A<56A8h) em compl 2
                  LDLI     R12,#2            ;
                  LDHI     R12,#0            ; carrega R12 com 0002
                  STMSK    #0C0H             ; gera condição de salto incondicional
                  JPRGR    R12               ; pula as duas próximas instruções
                  HALT                       ; nunca deve ser executada
                  HALT                       ; nunca deve ser executada
                  LDLI     R13,#testeJSRGR   ;
                  LDHI     R13,#testeJSRGR   ; carrega R13 com endereço de testeJSRGR
                  SUBI     R13,#testePC      ; obtém diferença entre valor do PC na hora
                                             ; da chamada e endereço da subrotina a chamar
                  STMSK    #03H              ; gera condição de salto incondicional
                  JSRGR    R13               ; salto incondicional para testeJSRGR
testePC: LDLI     R11,#testeJSRGR            ; rótulo testePC dá valor do PC na hora da chamada
                                             ; da linha acima desta (contendo JSRGR R13)
                  LDHI     R11,#testeJSRGR   ; carrega R11 com endereço de testeJSRGR
                  LDHI     R10,#80h          ; cria o número mais negativo possível em 16 bits,
                  LDLI     R10,#0            ;  ou seja, 8000H
                  SUBI     R10,#1            ; subtrai 1 dele para gerar V=1
                  STMSK    #1                ; máscara de salto se overflow reset
                  JPRG     R11               ; salto não deve ser executado
                  STMSK    #2                ; máscara de salto se overflow set
                  JSRG     R11               ; este salto salta!! Ao mesmo tempo testa aninhamento
                  POP      R3                ;  de subrotinas, para ver se pilha está OK!
                  POP      R2                ;
                  POP      R1                ; recupera o contexto
                  RTS                        ; final da subrotina
.endcode
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; area de dados
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
.data
n:                DB       #08H
end1:             db       #03H,#18H,#35H,#0ABH,#0CDH,#77H,#53H,#45H
end2:             db       #67H,#34H,#21H,#0BFH,#0FH,#0FDH,#11H,#01H
end3:             db       #00H              ; reservado para receber resultados
.enddata
```

Figura 6 - Programa exemplo para teste da arquitetura R12.

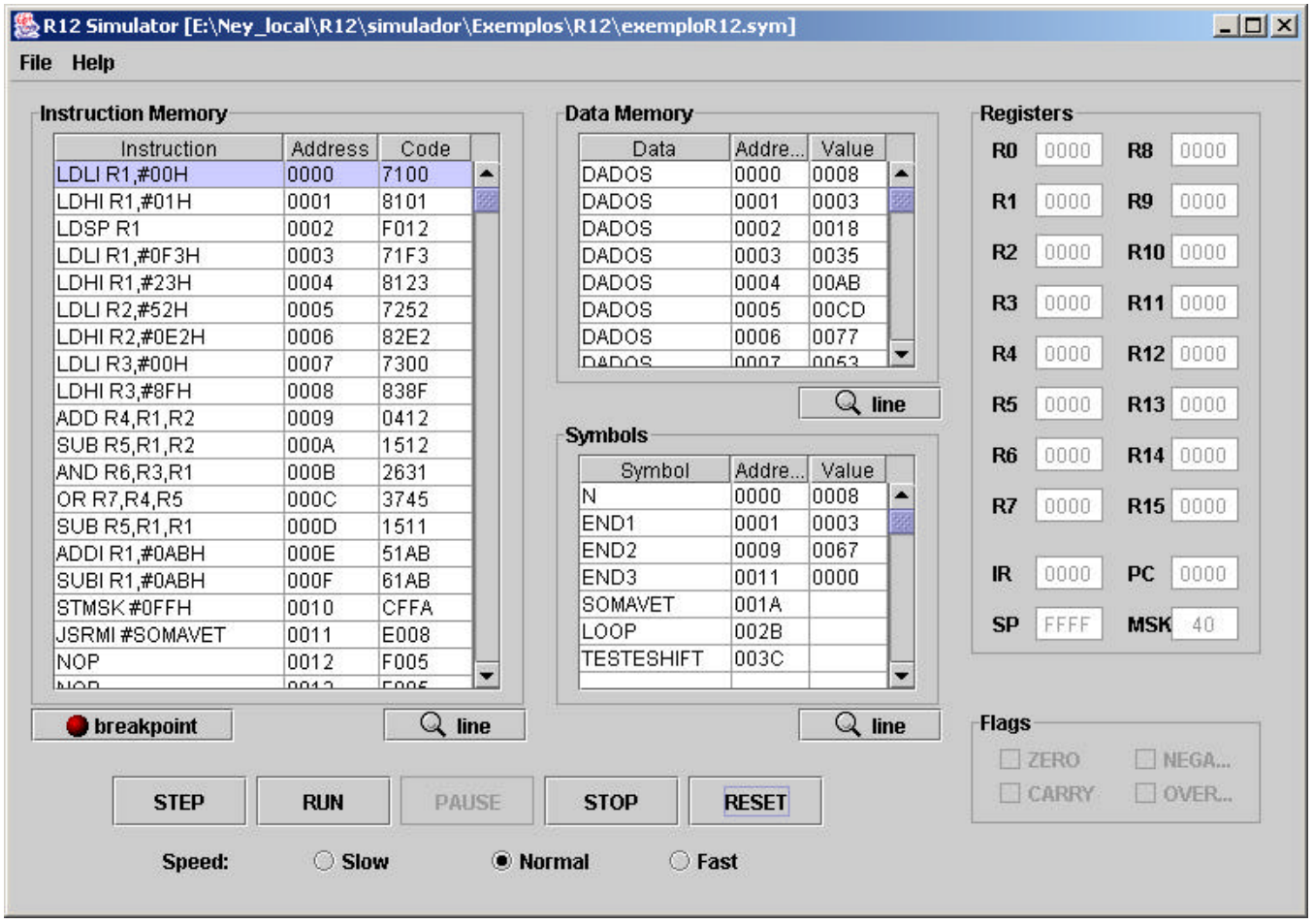


Figura 7 - Interface gráfica do montador/simulador do processador R12.