PROCESSADOR MULTI-CICLO - R9

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 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 teremos 2000 acessos à memória, realizando leitura e escrita a cada iteração do laço `for`. Caso tenhamos o valor de `i` armazenado em um registrador interno, apenas operamos 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 uma arquitetura *load-store* pode apresentar em relação a uma arquitetura baseada em acumulador.
- Dados e endereços nas arquiteturas Rx são de 16 bits. Logo, diz-se que a **palavra** do processador é de 16 bits, ou que se trata de um processador de 16 bits.
- O endereçamento de memória é a palavra, ou seja, cada endereço corresponde a um identificador de uma posição onde residem 16 bits ou 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. Comparar com a arquitetura Cleópatra. A instrução contém o código da operação e o(s) operando(s), caso exista(m).
- O bloco de controle é *hardwired*, e não microprogramado como na caso da Cleópatra.

Assim, este processador é praticamente uma máquina RISC, faltando contudo algumas características que existem em qualquer máquina 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 R9

- Instruções na R9 têm duração variável entre 2 e 4 ciclos de relógio. Logo, o CPI de qualquer organização não pipeline para a arquitetura R9 possui CPI entre 2 e 4. Comparar com a arquitetura Cleópatra.
- Não existem qualificadores de estado (*flags*) na R9. Condições como sinal de número em complemento de 2 e igualdade a 0 são verificadas via instruções lógicas que colocam resultados sobre registradores de uso geral. A condição de transbordo (*overflow*) aritmético é tratada como exceção e a detecção de vai-um não está diretamente disponível, podendo ser feita indiretamente, em software.
- Modos de endereçamento que implicam múltiplos acessos à memória (tais como o modo indireto) não existe. 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 R9.

3 A RELAÇÃO PROCESSADOR - MEMÓRIA - AMBIENTE

A Figura 1(a) ilustra a relação entre o processador, a memória externa e o mundo exterior ao subsistema processador-memória. O processador 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 três condições devem ser identificadas: processador parado devido à execução de uma instrução **HALT**, processador parado devido à tentativa de execução de uma instrução inválida e transbordo durante a execução de uma das instruções aritméticas. As duas primeiras exceções provocam uma parada (natural e forçada, respectivamente) do processador, enquanto a última consiste apenas de uma sinalização ativa durante o último ciclo de relógio da fase de execução da instrução que causou o transbordo. No caso de parada do processador, apenas um sinal externo de reset pode provocar o reinício da execução de instruções.

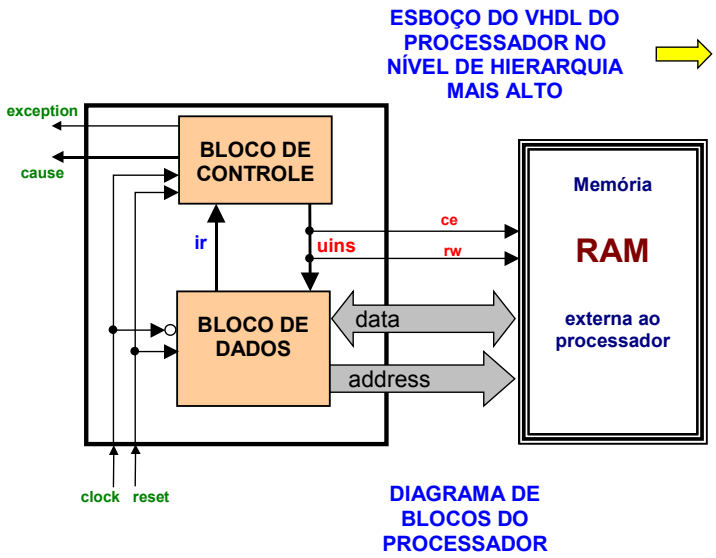
Os sinais providos pelo processador R9 para a troca de informações com a memória são: **data** (barramento bidirecional de 16 bits para os dados) e **address** (barramento de 16 bits contendo o endereço da posição de memória a ser acessada). O controle de acesso à memória é feito pelo bloco de controle, através dos sinais **ce** e **rw**. O sinal **ce** indica se está em curso uma operação com a memória (quando **ce** = 1) e o sinal **rw** indica se esta operação é de escrita (quando **rw** = 0) ou de leitura (quando **rw** = 1).

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 que serão executadas no bloco de dados a cada ciclo de relógio, ou seja as microoperações. Exemplos destas são cada uma das três seleções de registradores de leitura e escrita, a operação que a ULA executará e os sinais de controle de acesso à memória externa.

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

É importante ressaltar que os blocos de dados e 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 modifica os registradores. Com isto, sempre tem-se dados estáveis nas transições de **clock** em cada um dos blocos. 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).

A Figura 1(b) representa o nível mais alto da hierarquia do processador, através da linguagem VHDL. Nesta Figura, os blocos estão conectados entre si por *sinais*, sendo instanciados pelo comando *port map*.



(a) Diagrama de blocos processador-memória para a organização R9.

```
entity R9 is
    port( clock,reset: in std_logic;
          exception: out std_logic;
          cause: out std_logic_vector (1 downto 0);
          data: inout reg16;
          address: out reg16;
          ce,rw: out std_logic );
end R9;

architecture R9 of R9 is
      signal uins: microinstrucao;
      signal ir: reg16;
      component datapath ...
      component controlpath ...
   begin
      dp: datapath
          port map(ck=>clock, rst=>reset,
           uins=>uins, instrucao=>ir,
           address=>address, data=>data);

      ctrl: controlpath
            port map(ck=>clock, rst=>reset,
             exception=>exception, cause=>cause,
             uins=>uins, ir=>ir);
      ce <= uins.ce;
      rw <= uins.rw;
end R9;
```

(b) Descrição VHDL do nível mais alto da hierarquia de descrição do processador R9.

Figura 1 - Relação entre o processador, mundo exterior e a memória externa e esboço de descrição VHDL de topo.

4 CONJUNTO DE INSTRUÇÕES

A Tabela 1 descreve resumidamente cada instrução da arquitetura R9.

Tabela 1 – Mnemônicos, codificação e semântica resumida das instruções do processador R9.

Instrução	FORMATO DA INSTRUÇÃO				AÇÃO
	15 – 12	11 - 8	7 - 4	3 - 0	
ADD Rt, Rs1, Rs2	**0**	Rt	Rs1	Rs2	Rt ← Rs1 + Rs2;
SUB Rt, Rs1, Rs2	**1**	Rt	Rs1	Rs2	Rt ← Rs1 - Rs2;
AND Rt, Rs1, Rs2	**2**	Rt	Rs1	Rs2	Rt ← Rs1 and Rs2;
OR Rt, Rs1, Rs2	**3**	Rt	Rs1	Rs2	Rt ← Rs1 or Rs2;
XOR Rt, Rs1, Rs2	**4**	Rt	Rs1	Rs2	Rt ← Rs1 xor Rs2;
ADDI Rt, imed8	**5**	Rt	imed8		Rt ← Rt + ("00000000" & imed8);
SUBI Rt, imed8	**6**	Rt	imed8		Rt ← Rt - ("00000000" & imed8);
LDLI Rt, imed8	**7**	Rt	imed8		Rt ← Rt_high & imed8;
LDHI Rt, imed8	**8**	Rt	imed8		Rt ← imed8 & Rt_low;
LDRI Rt, imed4 (Rs2)	**9**	Rt	imed4	Rs2	Rt ← PMEM ((imed4 com ext. de sinal) +Rs2);
STRI Rt, imed4 (Rs2)	**A**	Rt	imed4	Rs2	PMEM ((imed4 com ext. de sinal) +Rs2) ← Rt;
JPR imed12	**B**	imed12			PC ← PC + (imed12 com ext. de sinal);
JALR imed12	**C**	imed12			R15← PC; PC ← PC + (imed12 com ext. de sinal);
SLT Rt, Rs1, Rs2	**D**	Rt	Rs1	Rs2	se Rs1< Rs2 então Rt ← 1, senão Rt ← 0;
MOV Rt, Rs1	**E**	Rt	Rs1	**0**	Rt ← Rs1;
INC Rt, Rs1	**E**	Rt	Rs1	**1**	Rt ← Rs1 + 1;
DEC Rt, Rs1	**E**	Rt	Rs1	**2**	Rt ← Rs1 - 1;
NOT Rt, Rs1	**E**	Rt	Rs1	**3**	Rt ← not (Rs1);
NEG Rt, Rs1	**E**	Rt	Rs1	**4**	Rt ← - Rs1;
SL0 Rt, imed4	**E**	Rt	imed4	**5**	Rt ← Rt deslocado imed4 bits à esq., com 0s à dir.;
SR0 Rt, imed4	**E**	Rt	imed4	**6**	Rt ← Rt deslocado imed4 bits à dir., com 0s à esq.;
SL1 Rt, imed4	**E**	Rt	imed4	**7**	Rt ← Rt deslocado imed4 bits à esq., com 1s à dir.;
SR1 Rt, imed4	**E**	Rt	imed4	**8**	Rt ← Rt deslocado imed4 bits à dir., com 1s à esq.;
RL Rt, imed4	**E**	Rt	imed4	**9**	Rt ← Rt rotacionado imed4 bits à esq.;
RR Rt, imed4	**E**	Rt	imed4	**A**	Rt ← Rt rotacionado imed4 bits à dir.;
INCI Rt, imed4	**E**	Rt	imed4	**B**	Rt ← Rt + ("000000000000" & imed4);
DECI Rt, imed4	**E**	Rt	imed4	**C**	Rt ← Rt - ("000000000000" & imed4);
JRG Rt	**E**	Rt	**0**	**F**	PC ← Rt;
SWI imed8	**F**	**0**	imed8		R15← PC; PC ← (0FFH) & imed8;
SKPEQ Rs1, Rs2	**F**	1	Rs1	Rs2	se Rs1 = Rs2 então PC ← PC + 1;
SKPNE Rs1, Rs2	**F**	2	Rs1	Rs2	se Rs1 ≠ Rs2 então PC ← PC + 1;
SKPEQI Rs2, imed4	**F**	3	imed4	Rs2	se (imed4 com sinal estendido) = Rs2, PC ← PC + 1;
SKPNEI Rs2, imed4	**F**	4	imed4	Rs2	se (imed4 com sinal estendido) ≠ Rs2, PC ← PC + 1;
NOP	**F**	**E**	**0**	**0**	nenhuma ação;
HALT	**F**	**F**	**0**	**0**	suspende seqüência de ciclos de busca e execução;

Convenções Utilizadas:

- **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 **PMEM(X)** representa o conteúdo de uma posição de memória cujo endereço é **X** (na leitura) ou a própria posição de memória (na escrita);
- Está implícito em todas as instruções o incremento do registrador PC após a fase de 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 **ADDI**, se **imed8** for 11111111 (-1 em complemento de 2, 8 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. 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 nas instruções de manipulação de bits (de **SL0** a **RL** na Tabela) e na instrução **SWI**;

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

- As **instruções aritméticas** são **ADD, ADDI, SUB, SUBI, NEG, INC, INCI, DEC DECI** e **SLT**;
- As **instruções lógicas** são **AND, OR, XOR** e **NOT**;
- As **instruções de manipulação de bits** são **SL0, SL1, SR0, SR1, RR** e **RL**;
- As **instruções de movimentação de dados** são **LDLI, LDHI, LDRI, STRI** e **MOV**;
- As **instruções de controle de fluxo de execução** são **JPR, JALR, JRG, SWI, SKPEQ, SKPNE, SKPEQI, SKPNEI e HALT**;
- Existe uma **instrução miscelânea**, **NOP**.

Algumas observações gerais e particulares sobre o conjunto de instruções são apresentadas abaixo:

- A arquitetura foi elaborada para privilegiar a simplicidade do conjunto de instruções sem 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 R9. 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**.
- Instruções de chamada e retorno de subrotina devem usar as instruções **JALR** e **JRG R15**, respectivamente. Note que o único registrador de uso geral diferenciado dos demais no processador R9 é o **R15**, usado implicitamente pela instrução **JALR**.
- A instrução **SLT** pode ser usada em conjunto com as instruções de prefixo **SKP** e com a instrução **JPR** para implementar comparações e controle de fluxo baseados em testes que não sejam de igualdade, visto que comparações de igualdade/desigualdade possuem instruções dedicadas. Importante: A comparação efetuada na instrução **SLT** é feita considerando os valores nos registradores como números representados em complemento de 2. **Tarefa 1**: Proponha uma seqüência de instruções do processador R9 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**).
- A instrução **SWI** (*software interrupt*) foi inserida na arquitetura para prover suporte a interrupções em software, empregadas em situações tais como chamadas de rotinas de entrada e saída padronizadas do sistema operacional. Elas funcionam exatamente como uma chamada de subrotina, e pressupõe a existência de uma tabela de saltos na região compreendendo as últimas 256 palavras do mapa de memória do processador (do endereço **FF00** ao **FFFF**). Logo, esta região não deve em princípio ser usada para conter dados ou instruções. **SWI** é uma instrução de controle de fluxo que usa uma forma limitada de modo de endereçamento absoluto. Para ser usada, cada uma destas posições deve conter o código de uma instrução de salto **JPR** para a posição de início da rotina de atendimento de interrupção. A outra instrução que usa endereçamento absoluto é **JRG**, um salto absoluto a registrador, necessária para uso em retorno de subrotinas. **Tarefa 2**: Discuta porquê estas instruções não afetam a relocabilidade de código objeto. **Tarefa 3**: Diga em que condições estas devem ser usadas para que a relocabilidade do código seja plena. Observação: Em implementações anteriores de arquiteturas Rx a instrução **SWI** não existia. Para aumentar a eficiência da simulação, muitas vezes a simulação era executada usando no testbench uma memória de menor tamanho que o previsto no mapa de memória do processador. Esta possibilidade não está disponível aqui, dado que as últimas 256 palavras do mapa são implicitamente usadas pela instrução **SWI** está em aberto até o momento a forma de contornar o problema de eficiência da simulação, sobretudo para aqueles que usem versões mais antigas do simulador Active-HDL (abaixo da versão 4).

5 REGISTRADORES DA ORGANIZAÇÃO R9 - BLOCO DE DADOS

O processador R9 conta com o seguinte conjunto de registradores de 16 bits:

- **IR** (*instruction register*): armazena o código de operação (*opcode*) da instrução atual e o(s) código(s) do(s) operando(s) desta.
- **PC** (*program counter*): é o contador de programa.
- 16 registradores de uso geral, **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 duas leituras simultâneas, colocando o conteúdo de um registrador no barramento de saída fonte 1 (**S1**) e o conteúdo de outro registrador (ou o mesmo) no barramento de saída fonte 2 (**S2**).

Há também registradores temporários, mostrados posteriormente, os quais são utilizados durante a execução das instruções. Os valores lidos do banco de registradores são armazenados nos registradores **RA** e **RB**. O valor obtido pela execução de uma dada operação lógico-aritmética é armazenado no registrador **RULA**.

6 EXECUÇÃO DAS INSTRUÇÕES NO BLOCO DE DADOS

A execução das instruções neste processador requer 2 a 4 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 ULA.** Comum a quase todas as instruções.
- **Ciclo 4 : acesso à memória e/ou atualização do banco registradores.** Conforme o tipo de operação realizada.

6.1 Ciclo de Busca da Instrução

- Busca a instrução endereçada pelo registrador **PC** na memória, grava esta no registrador **IR** e incrementa o **PC**:

IR ← PMEM(PC); PC++;

- A Figura 2 ilustra os componentes de hardware necessários para a execução do ciclo de busca da instrução. Deve-se notar que o registrador PC possui um hardware incrementador dedicado para si.

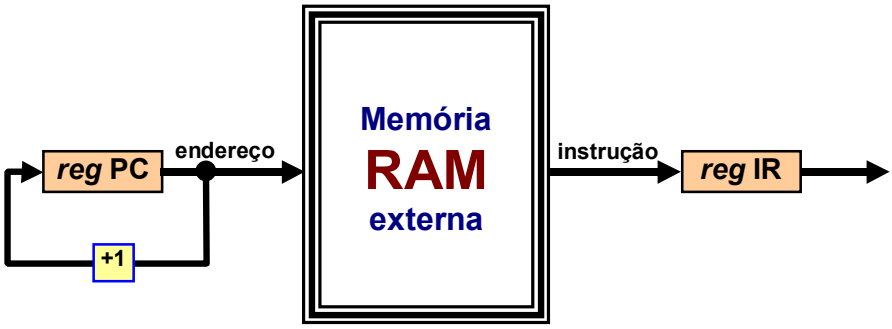


Figura 2 - Hardware necessário para executar o ciclo de busca.

6.2 Ciclo de Decodificação e Leitura de Registradores

- No segundo ciclo são lidos os valores dos operandos fonte das instruções de registradores de uso geral. O conteúdo dos registradores fonte são armazenados nos registradores **RA** e **RB**. Note que esta ação é feita em paralelo com a identificação da instrução particular. De fato, uma parte do trabalho de maioria das instruções (todas as que necessitam a leitura de pelo menos um valor de um dos registradores **Rt, Rs1** ou **Rs2**) está sendo adiantado aqui.
- A saída **S1** do banco de registradores é endereçada pelos bits 7 a 4 do registrador **IR**. Quando a operação envolve o registrador **Rt** usado como fonte, obtém-se **S1** a partir do endereçamento pelos bits 11 a 8. Exemplo: a instrução **ADDI**, onde **Rt** é somado a uma constante e armazena-se a soma neste mesmo registrador. Tal multiplexação reduz a necessidade de saídas do banco de registradores, diminuindo o tamanho do hardware.
- A saída **S2** do banco de registradores é endereçada pelos bits 3 a 0 do registrador **IR**.
- A Figura 3 ilustra os componentes de hardware para a leitura dos registradores fonte.

- Esta é efetivamente a última etapa das instruções **NOP** e **HALT**.
- A Tabela 2 define o conteúdo dos registradores **RA** e **RB** ao final do ciclo de decodificação e leitura de operandos para cada instrução. Deve-se notar que uma única instrução necessita **Rt** e **Rs2** em **RA** e **RB** respectivamente, a instrução **STRI**. Isto ocorre por dois motivos. O primeiro é acelerar o processamento da instrução, pois o dado estará em um registrador isolado, e não no banco de registradores no momento da escrita em memória. O segundo é facilitar a futura transformação da organização R9 em uma organização pipeline, assunto da disciplina de Arquitetura de Computadores I.

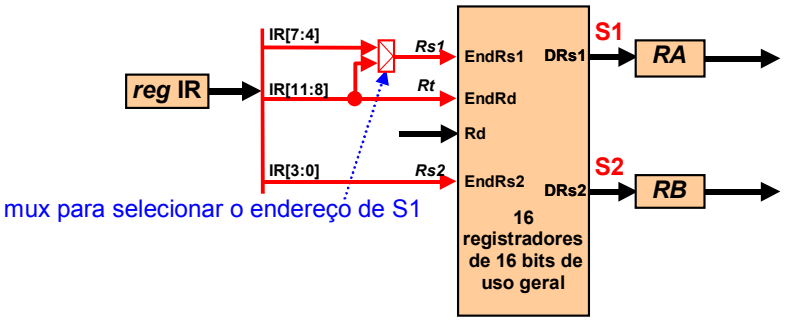


Figura 3 - Hardware para a leitura dos registradores fonte e escrita do registrador destino.

Tabela 2 – Conteúdos dos registradores RA e RB ao final do ciclo de decodificação, para cada instrução.

Instruções	RA	RB
ADD, SUB, AND, OR, XOR, SLT, SKPEQ, SKPNE	Rs1	Rs2
ADDI, SUBI, LDLI, LDHI, SL0, SR0, SL1, SR1, RL, RR, INCI, DECI, JRG	Rt	-
LDRI, SKPEQI, SKPNEI	-	Rs2
STRI	Rt	Rs2
MOV, INC, DEC, NOT, NEG,	Rs1	-
JPR, JALR, SWI, NOP, HALT	-	-

6.3 Ciclo de Operação com a ULA

- A ULA possui duas entradas e uma saída de dados. Além disto, a ULA possui uma entrada de controle para informar a operação a ser realizada em cada instante, e duas saídas de status de 1 bit, que informam respectivamente se houve transbordo aritmético em uma operação (sinal **ovf**) e se os operandos da ULA são iguais ou não (sinal **eq**).
- O ciclo de operação com a ULA é comum a quase todas as operações. Dada a variedade de instruções, necessita-se inserir multiplexadores nas entradas da ULA a fim de selecionar corretamente os operandos.
- O resultado da operação com a ULA é armazenado no registrador **RULA**. O sinal de transbordo aritmético é gerado a cada operação aritmética apenas. Para outras operações, ele deve ser mantido em 0.
- A Figura 4 ilustra os componentes de hardware para a operação com a ULA.

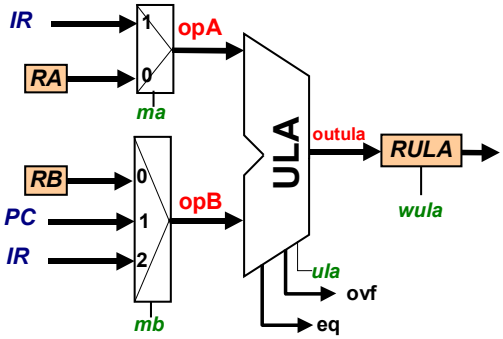


Figura 4 - Hardware para a operação com a ULA.

- A Tabela 3 define as entradas da ULA, **opA** e **opB**, conforme a instrução.

Tabela 3 – Entradas da ULA no início do ciclo de operação com a ULA para cada instrução.

Instruções	opA	opB
ADD, SUB, AND, OR, XOR, SLT, SKPEQ, SKPNE	RA	RB
ADDI, SUBI, LDLI, LDHI, SL0, SR0, SL1, SR1, RL, RR, INCI, DECI	RA	IR
LDRI, STRI, SKPEQI, SKPNEI	IR	RB
JPR, JALR	IR	PC
MOV, INC, DEC, NOT, NEG, JRG	RA	-
SWI	IR	-
NOP, HALT	-	-

6.4 Ciclos de Execução de Instruções

6.4.1 Execução das instruções lógicas e aritméticas com endereçamentos a registrador e imediato, das instruções de manipulação de bits e das instruções MOV, LDLI e LDHI.

- O quarto ciclo de relógio das instruções tratadas aqui grava o resultado do registrador **RULA** no banco de registradores, conforme o endereço do registrador destino. Este ciclo é chamado de *write-back*. A instrução **MOV** não precisa da ULA e gasta um ciclo a menos, usando o caminho que vai de **RA** de volta ao banco de registradores para escrita.
- A Figura 5 ilustra a organização de hardware para a execução dos casos de instrução tratados aqui.

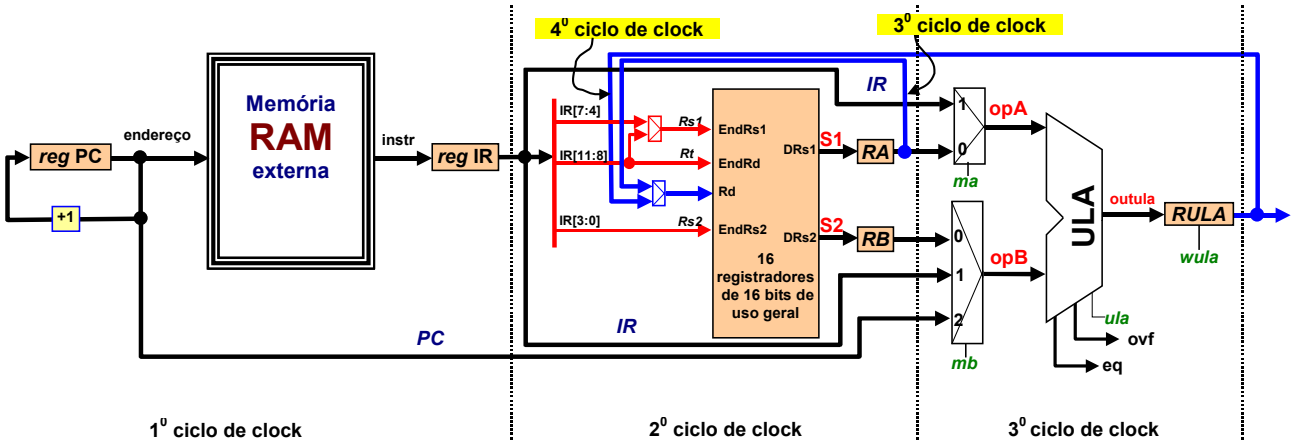


Figura 5 – Organização para a execução de instruções lógico-aritméticas em 4 ciclos de relógio.

- A instruções com modo de endereçamento imediato implicam em um dado registrador destino (em geral **Rt**) receber o resultado de uma operação entre este registrador e uma constante de 4 ou 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 um 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 devemos utilizar 2 instruções, **LDLI** e **LDHI**. Para ler/escrever um dado contido em um endereço qualquer de memória são necessárias 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 (**LDRI** ou **STRI**). Isto ocorre porque não há instruções que possam conter um endereço completo de memória (16 bits). Exemplo:

```
LDHI    R1, #03H
LDLI    R1, #27H ; armazena no registrador R1 o valor 0327H
LDRI    R5, 0(R1); armazena em R5 o conteúdo do endereço 0327H de memória
```

  - instruções de manipulação de bits (**SL0, SR0, SL1, SR1, RL, RR**): Usam **Rt** e um operando imediato que dá a quantidade de manipulação (deslocamento ou rotação) de bits. A diferença entre deslocamento e rotação é que no primeiro os bits de um dos extremos da palavra são perdidos, enquanto no segundo eles são ordenadamente inseridos no extremo oposto. Por exemplo, supor que o registrador

de uso geral **R3** contém o valor binário **1111000000000000**. O resultado de executar **SL0 R3, 2** é colocar **1100000000000000** em **R3**. Por outro lado, o resultado de executar **RL R3, 2** sobre o dado original é colocar **1100000000000011** em **R3**. O dado imediato é um valor de 4 bits, representando uma constante natural entre 0 a 15. **Importante**: Esta constante é uma das entradas da ULA na fase de execução, funcionando como informação de controle (a quantidade de bits a deslocar/rotacionar no registrador fonte/destino **Rt**). Deve-se notar que usar instruções com 0 é equivalente a uma instrução **NOP**;

- instruções de incremento/decremento imediato (**INCI, DECI**): somam ou subtraem uma constante curta ao registrador **Rt**, colocando o resultado de volta no mesmo registrador Como no caso das instruções **ADDI** e **SUBI**, estas não usam extensão de sinal.

6.4.2 Execução da instrução de leitura da memória (LDRI)

- O registrador *destino* (**Rt**) recebe o conteúdo da posição de memória endereçada pela soma do registrador fonte com uma constante entre –8 e +7, em complemento de 2: **Rt ← PMEM(imed4+Rs2)**. O registrador pode ser considerado como base e a constante o deslocamento (em inglês, *offset*) a partir da base.
- No quarto ciclo de relógio o registrador **RULA** endereça a memória, e o dado lido é gravado no banco de registradores, no registrador destino endereçado por **IR[11:8]**.
- Uma possível organização do bloco de dados para a execução da instrução **LDRI** aparece na Figura 6.

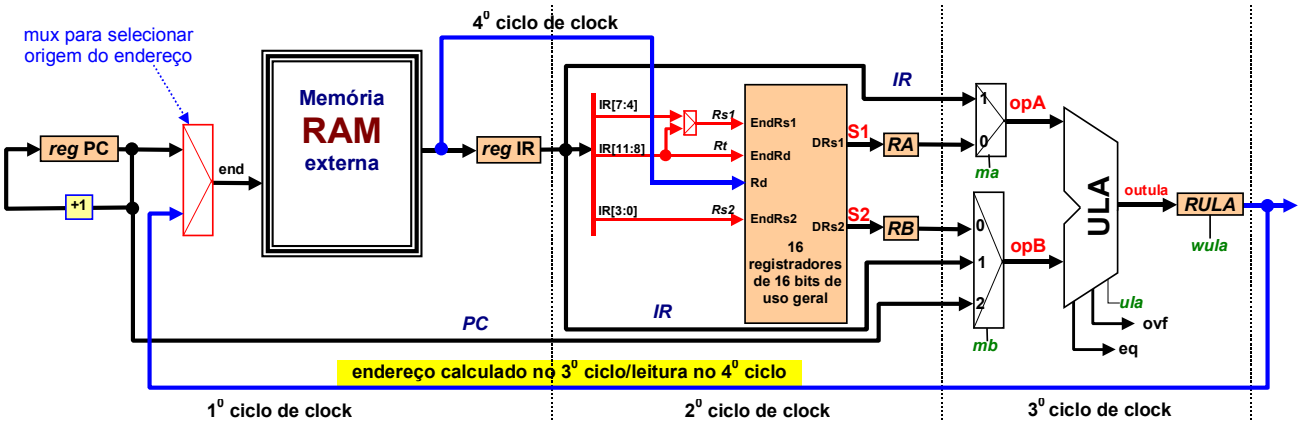


Figura 6 - Organização para a execução da instrução de leitura da memória.

6.4.3 Execução da instrução de escrita na memória (STRI)

- A posição de memória endereçada pela soma do conteúdo de um registrador fonte com uma constante entre –8 e +7, representada em complemento de 2 recebe o conteúdo de **Rt**: **PMEM(imed4+Rs2) ← Rt**.
- No segundo ciclo de relógio, o registrador endereçado por **IR[11:8] (Rt)** é lido para o registrador **RA**. No quarto ciclo, grava-se o conteúdo de **RA** no endereço definido por **RULA**.
- Uma possível organização para permitir a execução da instrução **STRI** é apresentada na Figura 7.

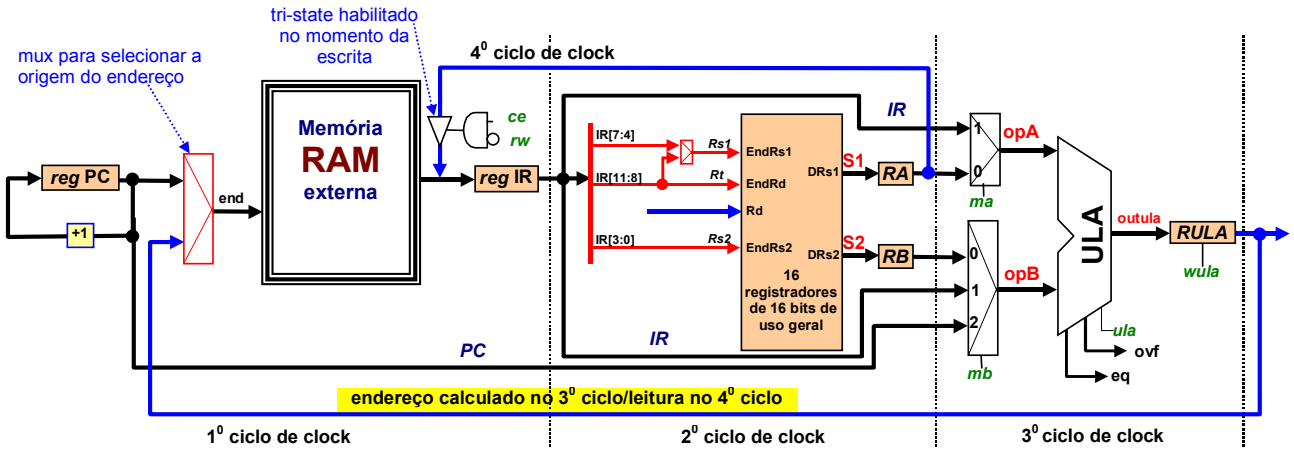


Figura 7 - Organização para a execução da instrução de escrita na memória.

6.4.4 Execução de instruções de controle de fluxo

- O endereço destino do salto é calculado, quando necessário, no terceiro ciclo de relógio, sendo este armazenado no registrador **RULA**. Caso seja um salto condicional (instruções com prefixo **SKP**, do inglês *skip*, significando pular por cima de) o valor do bit de status **eq** da ULA deve ser armazenado ou mantido (pela manutenção das entradas da ULA estáveis) no quarto ciclo, pois deve ser usado na execução condicional do salto. A instrução **JRG** não precisa da ULA e gasta um ciclo a menos, usando o caminho que vai de **RA** de volta ao banco de registradores para escrita.
- Caso o salto deva ser executado, o **PC** deve receber o conteúdo de **RULA**, ou ser incrementado **PC** instruções do tipo *skip*; Caso se trate das instruções **JALR** ou **SWI**, adicionalmente à escrita no **PC** o registrador **R15** deve ser carregado com o valor do **PC** incrementado após a fase de busca;
- Uma possível organização do bloco de dados para a execução dos saltos é apresentada na Figura 8.

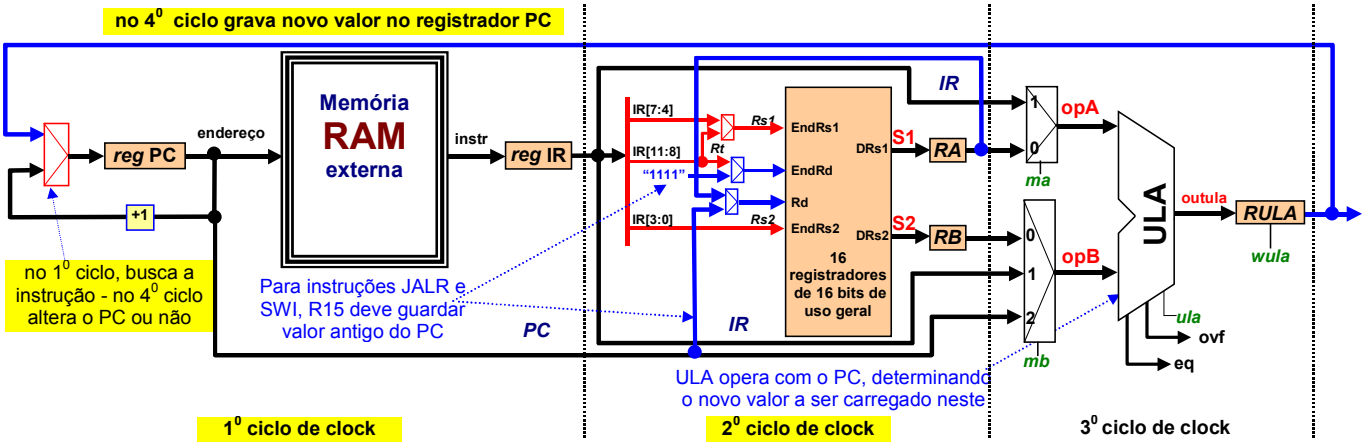


Figura 8 - Organização para a execução de instruções de controle de fluxo.

6.4.5 Uso de subrotinas na arquitetura R9

As instruções **JALR** (do inglês *Jump And Link Relative*, significando saltar e guardar o local de onde veio, ou seja manter uma ligação ou *link* com a origem do salto) e **JRG** (do inglês *Jump Register*, ou salto a registrador) existem para dar um suporte mínimo à chamada de subrotinas (procedimentos ou funções). Como na arquitetura Cleópatra, existe um registrador para implicitamente guardar o endereço de retorno. Na Cleópatra era o registrador **RS**, aqui, é o registrador de uso geral **R15**. No entanto, como a arquitetura Cleópatra, a R9 não dá suporte explícito para chamadas aninhadas de subrotinas (chamada a subrotina no corpo de uma subrotina), nem ao caso especial de aninhamento denominado recursão (ou seja, uma subrotina chamar a si própria). Este suporte deve ser implementado pelo software da aplicação. Para tanto, se pode reservar um registrador (digamos **R14**) para controlar o acesso a uma estrutura de dados em memória do tipo pilha para armazenar os endereços de retorno aninhados numa ordem e recuperá-los na ordem inversa. Vejamos um exemplo de trecho de programa para realizar isto, com um nível de aninhamento:

```
PROG:   LDHI    R14, #0FFH      ;
        LDLI    R14, #0         ; Carrega R14 com FF00H, fundo da pilha. Esta posição nunca é usada.
        JALR    ROT1            ; Salta para subrotina ROT1, guardando endereço de retorno em R15
...             ; Outras instruções do programa
        HALT    ; Fim do programa
ROT1:   DEC     R14, R14        ; Decrementa o apontador da pilha para primeira posição livre da pilha, FEFFH
        STRI    R15, #0(R14)    ; Coloca R15 na pilha
...             ; Outras instruções de ROT1
        JALR    ROT2            ; Salta para subrotina ROT2, guardando endereço de retorno em R15
...             ; Outras instruções de ROT1
        LDRI    R15, #0(R14)    ; Após voltar de ROT2 e antes de sair, recupera da pilha o endereço de retorno
        INC     R14, R14        ; Retira da pilha
        JRG     R15             ; volta ao programa PROG na posição imediatamente após a instrução JALR ROT1
ROT2:   ...                     ; instruções da subrotina ROT2
        JRG     R15             ; Como está na folha da hierarquia, não usa pilha, volta direto.
```

7 ORGANIZAÇÃO DO BLOCO DE DADOS

Unindo as diversas Figuras anteriores, obtemos o diagrama da Figura 9. O bloco de dados necessita **15** sinais de controle, organizados em 4 classes:

- habilitação de escrita em registradores (4): ***wPC***, ***wIR***, ***wAB***, ***wula***.
- controle de leitura/escrita na memória externa (2): ***ce*** e ***rw***.
- controle de multiplexadores (7): ***mpc*** (origem do PC), ***mAD*** (origem do endereço de memória), ***mRs1*** (origem do endereço para saída **S1** do banco de registradores), ***mRd*** (origem do endereço para entrada do banco de registradores), ***mDReg*** (origem dos dados para entrada do banco de registradores), ***ma*** (origem dos dados para o primeiro operando da ULA) e ***mb*** (origem dos dados para o segundo operando da ULA).
- a operação que a unidade lógica-aritmética executa (1): ***ula***.

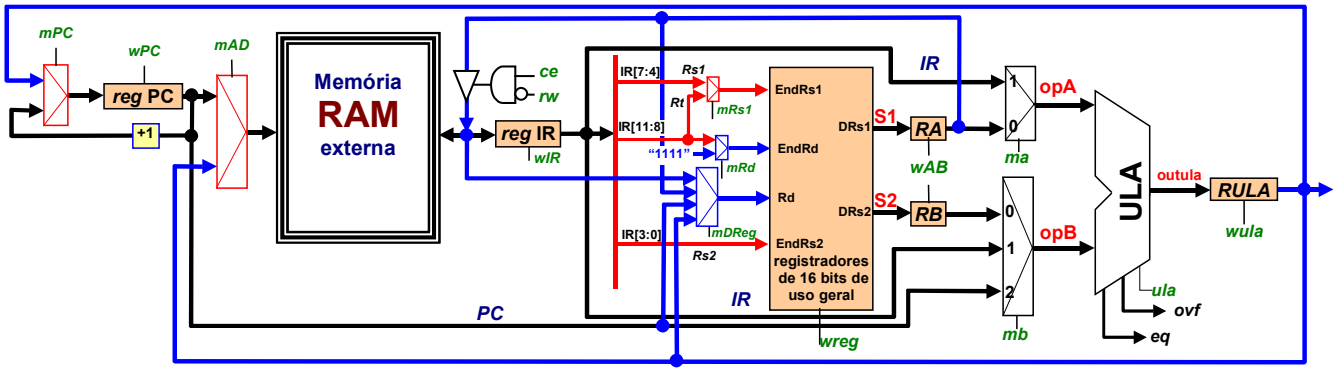


Figura 9 - Bloco de dados completo, com memória externa mostrada para fins de clareza. Nesta figura estão representados todos os 15 sinais que o bloco de controle deve gerenciar (em verde, itálico). Os sinais de clock e reset não estão representados, porém são utilizados por todos os registradores.

A Figura 10 ilustra a organização do banco de registradores, sob forma de um diagrama de blocos.

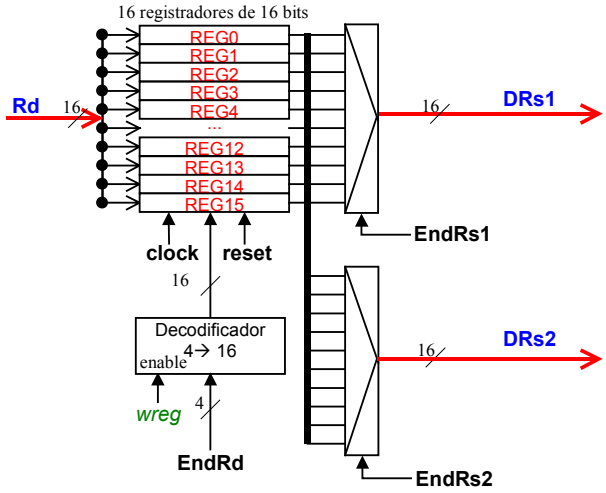


Figura 10 - Diagrama em blocos do banco de registradores de uso geral.

8 BLOCO DE CONTROLE

Para comandar a execução de instruções neste processador, é possível definir uma máquina de estados de controle. A Figura 11 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, assim como quando há acesso à memória (*leRAM* e *writeRAM*).

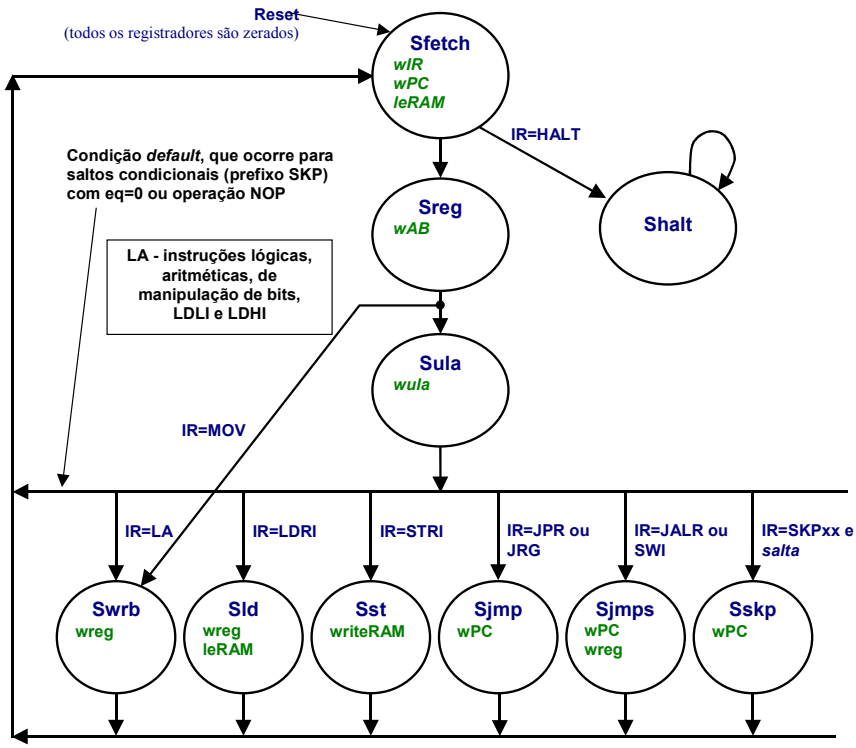


Figura 11 - Máquina de estados de controle para organização R9 proposta.

A função dos 10 diferentes estados é:

- **Sfetch:** primeiro ciclo de relógio, busca a instrução;
- **Sreg:** segundo ciclo de relógio, leitura dos registradores fonte;
- **Shalt:** segundo ciclo de relógio, finaliza a execução e aguarda reset;
- **Sula:** terceiro ciclo de relógio, operação com a ULA;
- **Swrb:** quarto ciclo de relógio, armazena resultado da operação da ULA no banco de registradores;
- **Sld:** quarto ciclo de relógio, busca dado da memória e armazena no banco de registradores;
- **Sst:** quarto ciclo de relógio, escreve conteúdo de registrador na memória;
- **Sjmp:** quarto ciclo de relógio, carrega o PC em saltos relativo imediato e a registrador;
- **Sjmps:** quarto ciclo de relógio, carrega o PC em saltos com salvamento do endereço de retorno;
- **Sskp:** quarto ciclo de relógio, instruções de salto condicional, que testa sinal salta e dependendo dele.

8.1 Decodificação das instruções / Definição da operação da ULA

A primeira ação a ser realizada no bloco de controle é a decodificação da instrução proveniente do bloco de dados. Como sugestão, pode-se definir em um *package* da descrição do processador um tipo enumerado contendo todas as possíveis instruções do processador R9. O código VHDL associado seria o seguinte:

```
type instrucao is
    ( add, sub, and_i, or_i, xor_i, addi, subi, ldli, ldhi, ldri, stri, jpr, jalr, slt, mov, inc,
      dec, not_i, neg, sl0, sl1, sr0, sr1, rl, rr, inci, deci, jrg, swi, skpeq, skpne, skpeqi,
      skpnei, nop, halt, 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 mencionada no início deste documento.

```
i <= add   when ir(15 downto 12)=0  else
     sub   when ir(15 downto 12)=1  else
     ……
     slt   when ir(15 downto 12)=13 else
     mov   when ir(15 downto 12)=14 and ir(3 downto 0)=0 else
     ……
     ……
     nop   when ir(15 downto 0)=x"FE00" else
     halt  when ir(15 downto 0)= x"FF00" else
     invalid_instruction;

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

8.2 Controle dos multiplexadores

Os sinais de controle dos multiplexadores (7) dependem do estado da máquina de controle (EA, de estado atual) e/ou da instrução corrente. Recomenda-se localizar os controles dos multiplexadores na Figura 9. Os sinais de controle são precedidos do sufixo "*uins.*", que designa a palavra de microinstrução. Abaixo mostra-se um exemplo de codificação VHDL para gerar estes sinais de controle.

Controle da origem dos dados para o PC (depende do estado de controle):

```
uins.mpc <= "1" when (EA=sjmp or EA=sjmps) else  -- nos saltos incondicionais, carrega PC
            "0";         -- por default realimenta o valor de PC incrementado de 1.
```

Resumindo, o bloco de controle é composto por três partes:

1) Máquina de estados que gera os sinais de controle de escrita/leitura na memória e escrita nos diversos registradores da arquitetura.
2) Decodificação de instruções.
3) Controle dos multiplexadores.

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 dará ênfase na 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 (no mínimo 40 instruções em linguagem de montagem) para o processador implementado, com utilização de pelo menos uma subrotina e pelo menos uma chamada aninhada de subrotina.
- O projeto deve ser composto de apenas 2 arquivos VHDL: um para o processador e outro para o *test_bench*. Mais que 2 arquivos VHDL entregues implica automaticamente a não avaliação do projeto. Arquivos adicionais, tais como códigos fonte e objeto do programa usado para validar a implementação não entram nesta conta.
- As apresentações ocorrerão na semana ***24-26/junho*** (2 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 15 dias antes da primeira apresentação, via sorteio entre os grupos restantes.
- Composição da nota (Observar que o peso da simulação é de 45%):

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

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 EXEMPLO PARA TESTAR TODAS AS INSTRUÇÕES DA R9

O código objeto abaixo 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.

```
;-------------------------------------------------------------------------------------------------------------------
                                    ;           PROGRAMA PRINCIPAL
        0000 7108    ; LDLI R1, #08H   ;
        0001 8110    ; LDHI R1, #10H   ; carrega R1 com 1008
        0002 7234    ; LDLI R2, #34H   ;
        0003 8212    ; LDHI R2, #12H   ; carrega R2 com 1234
        0004 7E00    ; LDLI R14, #0    ;
        0005 8E01    ; LDHI R14, #01   ; carrega R14 com 0100 - será usado como fundo da pilha
        0006 0412    ; ADD  R4, R1, R2 ; soma - resultado em R4: 223C
        0007 1512    ; SUB  R5, R1, R2 ; subtrai - resultado em R5: FDD4
        0008 2612    ; AND  R6, R1, R2 ; and - resultado em R6: 1000
        0009 3712    ; OR   R7, R1, R2 ; or - resultado em R7: 123C
        000A 4812    ; XOR  R8, R1, R2 ; xor - resultado em R8: 023C
        000B 5101    ; ADDI R1, #01    ; soma imediata com 01 - resultado em R1: 1009
        000C 510F    ; ADDI R1, #0FH   ; soma imediata com 0F - resultado em R1: 1018
        000D 51FF    ; ADDI R1, #0FFH  ; soma imediata com FF - resultado em R1: 1117
        000E 6201    ; SUBI R2, #01    ; subtração imediata de 01 - resultado em R2: 1233
        000F 6204    ; SUBI R2, #04    ; subtração imediata de 04 - resultado em R2: 122F
        0010 62FF    ; SUBI R2, #0FFH  ; subtração imediata de FF - resultado em R2: 1130
        0011 C0EE    ; JALR #0EEH      ; salta para subrotina ROT1, no endereço 0100 R15 recebe 0012 (end. retorno)
        0012 FE00    ; NOP             ; ao voltar de ROT1, não faz nada, apenas para testar NOP
        0013 73DC    ; LDLI R3, #0DCH  ;
        0014 83FE    ; LDHI R3, #0FEH  ; carrega R3 com FEDC
        0015 B015    ; JPR  #15H       ; salto relativo incondicional para o endereço 002B
        0016 FF00    ; HALT            ; esta linha nunca deve ser executada
        002B D012    ; SLT  R0, R1, R2 ; carrega R0 com 1, pois 1117 < 1130
        002C F101    ; SKPEQ  R0, R1   ; não se deve executar skip, pois 1 \= 1117
        002D F200    ; SKPNE  R0, R0   ; não se deve executar skip, pois 1 = 1
        002E F310    ; SKPEQI R0, #1   ; deve saltar a próxima instrução (executar skip), pois 1 = 1
        002F FF00    ; HALT            ; esta linha nunca deve ser executada
        0030 F480    ; SKPNEI R0, #8   ; deve saltar a próxima instrução, pois 1 \= 8
        0031 FF00    ; HALT            ; esta linha nunca deve ser executada
        0032 E501    ; INC  R5, R0     ; carrega R5 com 2
        0033 E552    ; DEC  R5, R5     ; carrega R5 com 1
        0034 E653    ; NOT  R6, R5     ; carrega R6 com FFFE
        0035 E664    ; NEG  R6, R6     ; carrega R6 com 2
        0036 FF00    ; HALT            ; programa deve parar de executar aqui
                                    ;           FIM DO PROGRAMA PRINCIPAL
;-------------------------------------------------------------------------------------------------------------------
        0037 EF0F    ; JRG  R15        ; Atendimento de interrupção de software, só retorna
;-------------------------------------------------------------------------------------------------------------------
                                    ;           ROT1 - Rotina chamada pelo programa principal
        0100 EEE2    ; DEC  R14, R14   ; Decrementa apontador de pilha - resultado em R14: 00FF
        0101 AF0E    ; STRI R15,#0(R14); Salva conteúdo de R15, 0012 na pilha, na posição 00FF de memória
        0102 EA70    ; MOV  R10, R7    ; carrega R10 com 123C
        0103 EA45    ; SL0  R10, #4    ; Testes de deslocamento e rotações carrega R10 com 23C0
        0104 EA37    ; SL1  R10, #3    ; carrega R10 com 1E07
        0105 EA86    ; SR0  R10, #8    ; carrega R10 com 001E
        0106 EA28    ; SR1  R10, #2    ; carrega R10 com C007
        0107 EAB9    ; RL   R10, #0BH  ; carrega R10 com 3E00
        0108 EA5A    ; RR   R10, #5    ; carrega R10 com 01F0
        0109 C006    ; JALR #6         ; salta p/ subrotina ROT2, no endereço 0110. R15 recebe 010A (end. retorno)
        010A F01A    ; SWI  #1AH       ; teste de SWI. R15 recebe 010B e salta para posição FF1A
        010B 9F0E    ; LDRI R15,#0(R14); ao voltar de SWI, recupera endereço de retorno (0012) da pilha
        010C EEE1    ; INC  R14, R14   ; incrementa apontador de pilha, retirando dado virtualmente desta
        010D EF0F    ; JRG  R15        ; retorna de ROT1 para o programa principal na posição 0012
;-------------------------------------------------------------------------------------------------------------------
                                    ;           ROT2 - Rotina chamada pela rotina ROT1
        0110 DC11    ; SLT  R12, R1, R1; teste para SLT colocar 0 em R12 (posição 002B testou colocar 1 em R0)
        0111 F30C    ; SKPEQI R12, #0  ; skip deve ocorrer. Como subrotina folha, não pode executar JALR nem SWI.
        0112 FF00    ; HALT            ; esta linha nunca deve ser executada
        0113 EF0F    ; JRG  R15        ; retorna para ROT1
;-------------------------------------------------------------------------------------------------------------------
                                    ;           TABELA de SALTOS PARA INSTRUÇÃO SWI
        FF19 FF00    ; HALT            ; tabela de saltos - posição que não deve ser usada
        FF1A B11C    ; JPR  #11CH      ; tabela de saltos - posição a ser usada salta para posição 0037
        FF1B FF00    ; HALT            ; tabela de saltos - posição que não deve ser usada
;-------------------------------------------------------------------------------------------------------------------
```

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 12 mostra a janela do simulador. A esquerda desta figura está apresentada a tabela de memória, contendo em cada linha a instrução em *assembly*, o endereço da posição da memória e o código objeto. Ao centro

é inserida a tabela de símbolos, onde são apresentados o nome do símbolo, 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 *Lento*, *Normal* e *Rápido*. Os qualificadores de estado encontram-se na parte inferior à direita.

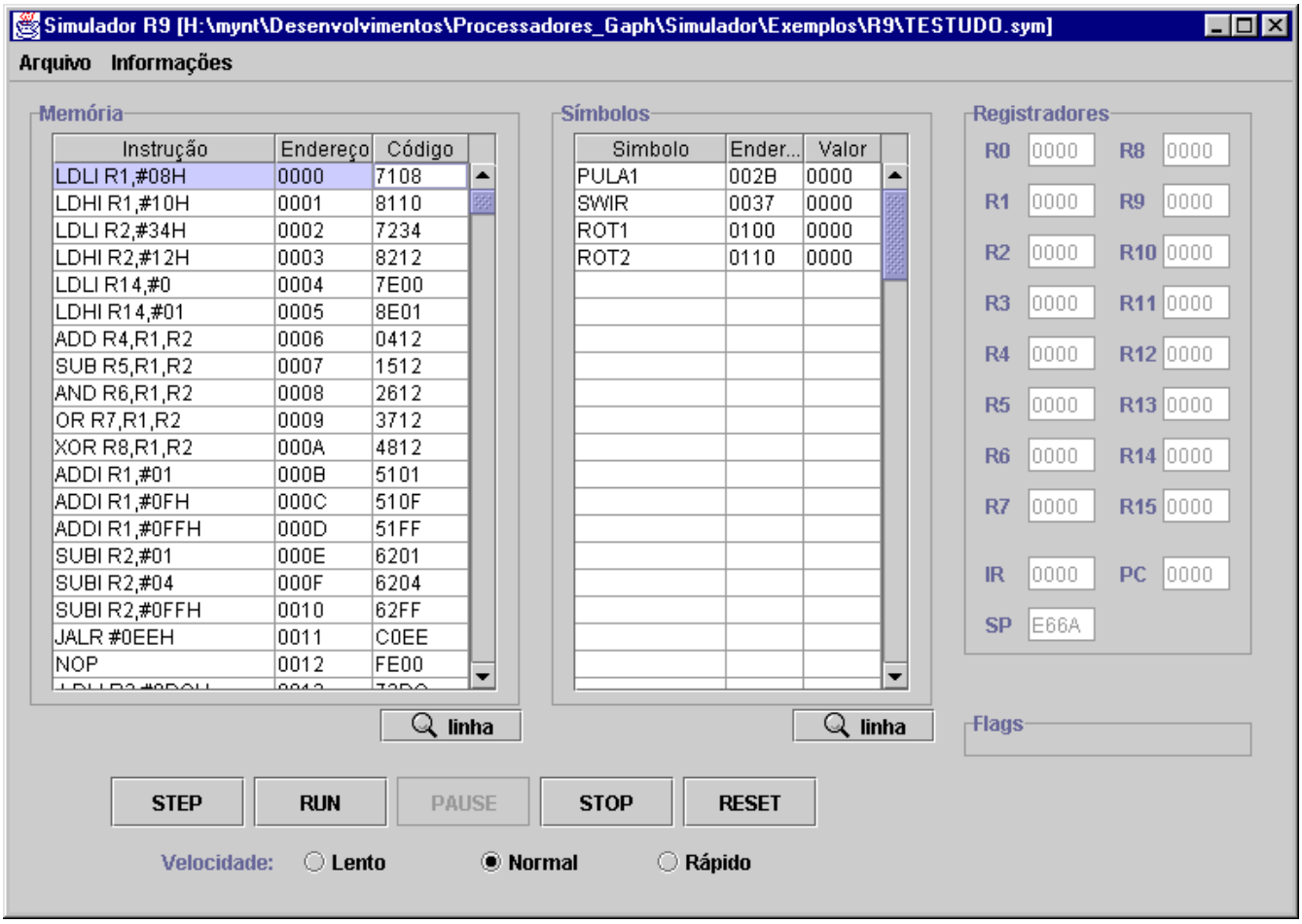


Figura 12 - Interface gráfica do montador/simulador do processador R9.

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:

- <file>.hex – para download na placa de prototipação;
- <file>.sym – para uso do simulador;
- <file>.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 chamada do simulador. 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 afim 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.