Última alteração: 19/Março/2013 por Everton Alceu Carara

PROCESSADOR MULTI-CICLO - R8

1 CARACTERÍSTICAS GERAIS

- Arquitetura load-store: as operações lógico/aritméticas são executadas entre registradores, e as operações de acesso à memória só executam ou uma leitura (*load*) ou uma escrita (*store*).
- Banco de registradores: devido à característica *load/store*, o processador deve ter um conjunto grande de registradores, para reduzir o número de acessos à memória (em processadores reais este acesso externo representa perda de desempenho). 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 registrador(es) especial (ais), denominado(s) acumulador(es). 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. Caso tenhamos o valor de 'i' armazenado em registrador, apenas operamos sobre o registrador, sem acessos à memória externa durante a maior parte do tempo!
- Formato regular para as instruções: todas as instruções possuem exatamente o mesmo tamanho, e ocupam 1 palavra de memória. A instrução contém o código da operação e o(s) operando(s), caso exista(m).
- Poucos modos de endereçamento.
- Bloco de controle *hardwired*, e não micro-programado.

Assim, este processador é praticamente uma máquina RISC, faltando contudo algumas características que existem em qualquer máquina RISC, tal como *pipelines*.

Características específicas deste processador multi-ciclo:

- dados e endereços são de 16 bits (processador de 16 bits).
- endereçamento de memória a palavra (cada endereço corresponde ao identificador de uma posição de memória onde residem 16 bits de conteúdo).
- banco de registradores com 16 registradores de uso geral.
- 4 *flags* de estado: negativo (n), zero (z), carry (c), overflow (v).
- execução das instruções em 3 ou 4 ciclos, ou seja, CPI (*Cycles Per Instruction*) entre 3 e 4.

2 CONJUNTO DE INSTRUÇÕES

O conjunto de instruções do processador realiza as seguintes operações:

- Operações lógicas e aritméticas binárias (com 2 operandos): soma, subtração, AND, OR e XOR.
- Operações lógicas e aritméticas com constantes curtas (8 bits): soma, subtração.
- Operações unárias (com 1 operando): deslocamento para direita ou esquerda e inversão (NOT).
- Carga de metade de um registrador com uma constante curta (LDL e LDH).
- Inicialização do apontador de pilha (LDSP) e retorno de subrotina (RTS).
- NOP (*no operation*): operação vazia (útil para laços de espera e reserva de espaço).
- HALT: suspende a execução de instruções.
- *Load*: leitura de posição de memória para um registrador (LD).
- *Store*: armazenamento de dado de um registrador em uma posição de memória (ST).
- Saltos e chamada de subrotina com endereçamento *relativo* com deslocamento curto ou longo (contido em um registrador) e endereçamento *absoluto* (a registrador).
- Inserção e remoção de valores no/do topo da pilha (PUSH e POP).

TABELA DE INSTRUÇÕES DO PROCESSADOR R8:

Instrução	FORMATO DA INSTRUÇÃO				
	15 – 12	11 - 8	7 - 4	3 - 0	AÇÃO ; *flags*
ADD Rt, Rs1,Rs2	**0**	R *target*	R source1	R source2	Rt ← Rs1 + Rs2; *wnz*; *wcv*
SUB Rt, Rs1,Rs2	**1**	R *target*	R source1	R source2	Rt ← Rs1 - Rs2; *wnz*; *wcv*
AND Rt, Rs1,Rs2	**2**	R *target*	R source1	R source2	Rt ← Rs1 *and* Rs2; *wnz*
OR Rt, Rs1,Rs2	**3**	R *target*	R source1	R source2	Rt ← Rs1 *or* Rs2; *wnz*
XOR Rt, Rs1,Rs2	**4**	R *target*	R source1	R source2	Rt ← Rs1 *xor* Rs2; *wnz*
ADDI Rt, *cte8*	**5**	R *target*	Constante (*unsigned*)		Rt ← Rt + ("00000000" & constante); *wnz*; *wcv*
SUBI Rt, *cte8*	**6**	R *target*	Constante (*unsigned*)		Rt ← Rt - ("00000000" & constante); *wnz*; *wcv*
LDL Rt, *cte8*	**7**	R *target*	Constante		Rt ← RtH & constante
LDH Rt, *cte8*	**8**	R *target*	Constante		Rt ← constante & RtL
LD Rt, Rs1,Rs2	**9**	R *target*	R source1	R source2	Rt ← PMEM (Rs1+Rs2)
ST Rt, Rs1,Rs2	**A**	R *target*	R source1	R source2	PMEM (Rs1+Rs2) ← Rt
SL0 Rt, Rs1	**B**	R *target*	R source1	**0**	Rt ← Rs1[14:0] & 0 ; *wnz*
SL1 Rt, Rs1	**B**	R *target*	R source1	**1**	Rt ← Rs1[14:0] & 1 ; *wnz*
SR0 Rt, Rs1	**B**	R *target*	R source1	**2**	Rt ← 0 & Rs1 [15:1]; *wnz*
SR1 Rt, Rs1	**B**	R *target*	R source1	**3**	Rt ← 1 & Rs1 [15:1]; *wnz*
NOT Rt, Rs1	**B**	R *target*	R source1	**4**	Rt ← not (Rs1) ; *wnz*
NOP	**B**	-	-	**5**	nenhuma ação
HALT	**B**	-	-	**6**	suspende seqüência de ciclos de busca e execução
LDSP Rs1	**B**	-	R source1	**7**	SP ← Rs1 (inicializa o apontador de pilha)
RTS	**B**	-	-	**8**	PC ←PMEM(SP+1) ; SP←SP+1
POP Rt	**B**	R *target*	-	**9**	Rt ←PMEM(SP+1) ; SP←SP+1
PUSH Rt	**B**	R *target*	-	**A**	PMEM(SP)←Rt ; SP←SP-1
JMPR Rs1	**C**	-	R source1	**0**	PC ← PC + Rs1 (não depende de flag de estado)
JMPNR Rs1	**C**	-	R source1	**1**	if (*n* =1) PC ← PC + Rs1
JMPZR Rs1	**C**	-	R source1	**2**	if (*z* =1) PC ← PC + Rs1
JMPCR Rs1	**C**	-	R source1	**3**	if (*c* =1) PC ← PC + Rs1
JMPVR Rs1	**C**	-	R source1	**4**	if (*v* =1) PC ← PC + Rs1
JMP Rs1	**C**	-	R source1	**5**	PC ← Rs1 (não depende de flag de estado)
JMPN Rs1	**C**	-	R source1	**6**	if (*n* =1) PC ← Rs1
JMPZ Rs1	**C**	-	R source1	**7**	if (*z* =1) PC ← Rs1
JMPC Rs1	**C**	-	R source1	**8**	if (*c* =1) PC ← Rs1
JMPV Rs1	**C**	-	R source1	**9**	if (*v* =1) PC ← Rs1
JSRR Rs1	**C**	-	R source1	**A**	PMEM(SP)←PC ; SP←SP-1; PC← PC+ Rs1
JSR Rs1	**C**	-	R source1	**B**	PMEM(SP)←PC ; SP←SP-1; PC← Rs1
JMPD desloc	**D**	**-**	Deslocamento (10 bits)		PC ← PC + **ext_sinal &** desloc
JMPND desloc	**E**	**0**	Deslocamento (10 bits)		if (*n* =1) PC ← PC + **ext_sinal &** desloc
JMPZD desloc	**E**	**1**	Deslocamento (10 bits)		if (z=1) PC ← PC + **ext_sinal &** desloc
JMPCD desloc	**E**	**2**	Deslocamento (10 bits)		if (*c* =1) PC ← PC + **ext_sinal &** desloc
JMPVD desloc	**E**	**3**	Deslocamento (10 bits)		if (*v* =1) PC ← PC + **ext_sinal &** desloc
JSRD desloc	**F**	Deslocamento (12 bits)			PMEM(SP)←PC ; SP←SP-1; PC← PC + **ext_sinal &** desloc

- As seguintes convenções foram utilizadas na tabela:

RtH:	**8 bits mais significativos de Rt**
RtL:	**8 bits menos significativos de Rt**
&:	**concatenação de vetores de bits**
←:	**atribuição de valor a registrador ou posição de memória**
PMEM(x):	**conteúdo de posição de memória cujo endereço é x**
Rt :	**R*target* [destino]**
Rs1 :	**Rsource1**
Rs2 :	**Rsource2**
***wnz*:**	**ativam o armazenamento dos *flags* de estado negativo e zero**
***wcv*:**	**ativam o armazenamento dos *flags* de estado *carry* e *overflow***

3 REGISTRADORES DO BLOCO DE DADOS

O processador 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) operando(s) desta.
- PC (*program counter*): é o contador de programa.
- SP (*stack pointer*): armazena o endereço do topo da pilha, controla a chamada e retorno de subrotinas. Deve ser inicializado a cada programa com a instrução LDSP (carrega endereço do topo da pilha).
- 16 registradores de propósito 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 *SOURCE1* (S1) e o conteúdo de outro registrador (ou o mesmo) no barramento de saída *SOURCE2* (S2).
- 4 *bits* (*flags*) de estado, denominados ***n*** (negativo)***, z*** (zero)***, c*** (carry) e ***v*** (overflow), utilizados para controle dos saltos e chamadas a subrotinas. O estado dos *flags*, 0 ou 1, é determinado durante as operações lógicas/aritméticas.

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*. O programador não tem acesso a estes registradores.

4 RELAÇÃO ENTRE O PROCESSADOR E A MEMÓRIA EXTERNA

A Figura 1a ilustra a relação entre o processador e a memória externa. O processador recebe do mundo externo dois sinais de controle: *clock*, que sincroniza os eventos internos ao processador; e *reset*, que inicializa o processador para iniciar a execução de instruções a partir do endereço zero da memória.

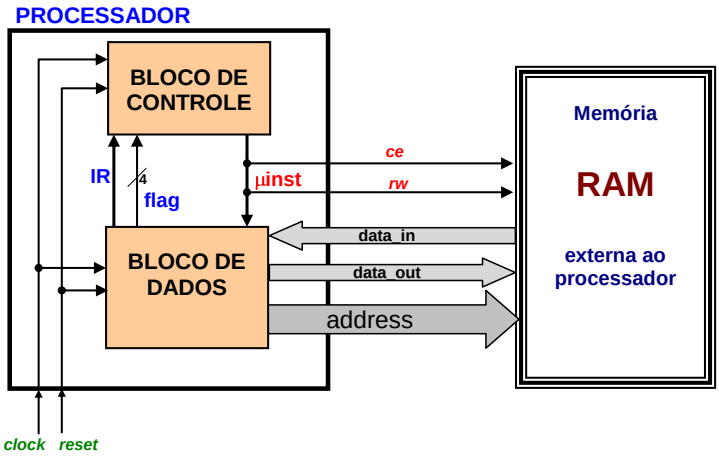
O bloco de controle (*control path*) gera a microinstrução (μinst) para a execução das instruções. A microinstrução é responsável por comandar as ações que serão executadas no bloco de dados (*data path*), como seleção de registradores, operação que a ULA executará e acesso à memória externa.

O bloco de dados envia para o bloco de controle a instrução corrente (conteúdo do ***IR***) e os qualificadores de estado (***flags***). O bloco de dados também é responsável pela comunicação com a memória externa. Os sinais para a troca de informações com a memória são: *data_in* (barramento de 16 bits para leitura de dados), *data_out* (barramento de 16 bits para escrita de dados) e *address* (barramento de 16 bits para endereçar a memória).

O controle de acesso à memória é feito pelo bloco de controle, através dos sinais *ce* e *rw*. O sinal *ce* (*ce = 1*) indica se está em curso uma operação com a memória e o sinal *rw* indica se esta operação é de escrita (*rw = 0*) ou de leitura (*rw = 1*).

Os blocos de dados e controle operam em fase (mesma borda do sinal de clock). Em uma borda do clock (por exemplo, subida) o bloco de controle gera a micro-instrução, e na borda seguinte (subida) o bloco de dados modifica os registradores.

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



(a) Diagrama de blocos processador-memória

PROCESSADOR

```
entity R8 is
      port( clk,rst: in std_logic;
            data_in: in std_logic_vector(15 downto 0);
            data_out: out std_logic_vector(15 downto 0);
            address: out std_logic_vector(15 downto 0);
            ce,rw: out std_logic );
end R8;

architecture structural of processador is
      signal flag: std_logic_vector(3 downto 0);
      signal uins: microinstrucao;
      signal ir: std_logic_vector(3 downto 0);
begin
      DATA_PATH: DataPath
          port map( uins=>uins, clk=>clk, rst=>rst,
                    instrucao=>ir,endereco=>address,
                    data_in=>data_in, data_out=>data_out,
                     flag=>flag);
      CONTROL_PATH: ControlPath
            port map(uins=>uins, clk=>clk, rst=>rst,
                      flag=>flag, ir=>ir);
      ce <= uins.ce;
      rw <= uins.rw;

end structural;
```

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

Figura 1 - Relação entre o processador e a memória externa.

5 EXECUÇÃO DAS INSTRUÇÕES NO BLOCO DE DADOS

A execução das instruções neste processador requer 3 ou 4 ciclos de clock (a única exceção é a instrução *halt*, que é executada em dois ciclos).

Os ciclos são assim denominados:

- **Ciclo 1 : busca da instrução.** Comum a todas as instruções.
- **Ciclo 2 : decodificação da instrução/leitura de registradores.** Comum a todas as instruções, exceto o *halt*.
- **Ciclo 3 : operação com a ula.** Comum a todas as instruções, exceto o *halt*.
- **Ciclo 4 : execução.** Conforme o tipo de operação realizada.

5.1 Ciclo de Busca da Instrução

- Busca a instrução endereçada pelo registrador *PC* na memória, grava a instrução 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.

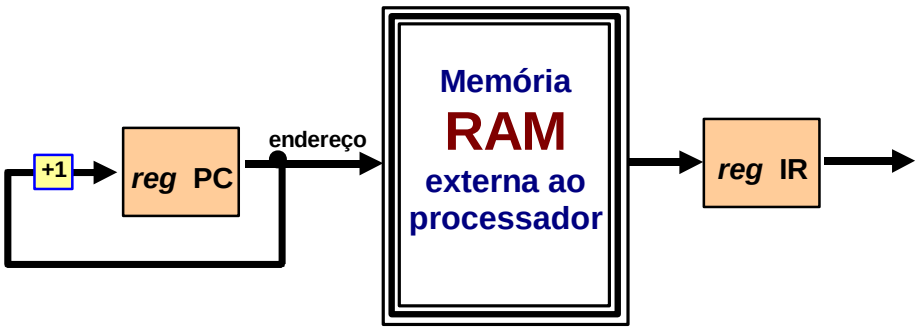


Figura 2 - Hardware para executar a busca.

5.2 Ciclo de Decodificação da instrução/Leitura de Registradores

- No segundo ciclo é verificada a instrução buscada e são lidos os registradores que são operandos (fontes) para as instruções, **independentemente se a instrução usa ou não os dois registradores**, *source1* (S1) e *source2* (S2). Os registradores fonte são armazenados nos registradores *RA* e *RB*.
- O registrador *source1* é endereçado pelos bits 7 a 4 do registrador IR.
- O registrador *source2* pode ser endereçado ou pelos bits 3 a 0 ou pelos bits 11 a 8 do registrador IR. Quando a operação envolve o registrador destino (*target*) como fonte, endereça-se o *source2* pelos bits 11 a 8. Exemplo: ADDI, onde é somado a um dado registrador uma constante e armazena-se a soma neste mesmo registrador.
- A Figura 3 ilustra os componentes de hardware necessários para a leitura dos registradores fonte.

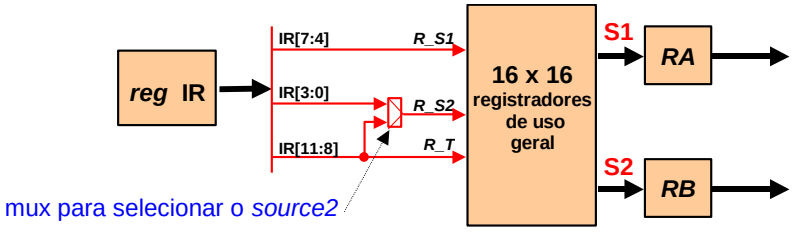


Figura 3 - Hardware para a leitura dos registradores fonte.

5.3 Ciclo de Operação com a ULA

- O ciclo de operação com a ULA também é comum a todas as operações (exceto instrução *halt*). 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*, e conforme a operação executada, armazena-se os *flags* de estado (n, z, c, v).
- A Figura 4 ilustra os componentes de hardware necessários para a operação com a ULA.

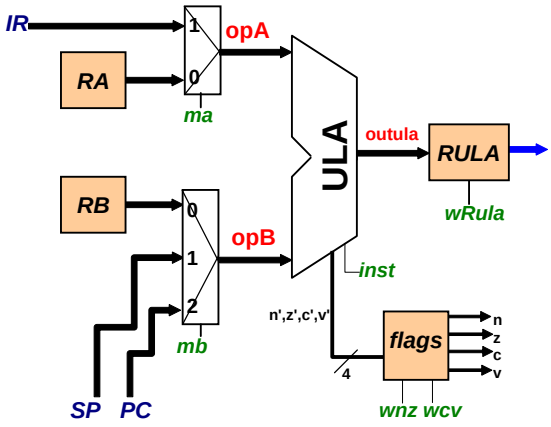


Figura 4 - Hardware para a operação com a ULA.

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

Instruções	opA	opB
ADD, SUB, AND, OR, XOR, LD, ST, SHIFTS, NOT, LDSP	RA	RB
ADDI, SUBI, LDL, LDH	IR	RB
RTS, POP	-	SP
Saltos e chamadas de sub-rotinas relativos ao PC	RA	PC
Saltos e chamadas de sub-rotinas absolutos	RA	-
Saltos e chamadas de sub-rotinas com deslocamento curto	IR	PC

5.4 Ciclo de Execução da Instrução

5.4.1 Execução das instruções lógico-aritméticas e endereçamento imediato.

- O quarto ciclo de clock das operações lógico-aritméticas grava o resultado do registrador *RULA* no banco de registradores, conforme o endereço do registrador destino. Este ciclo é chamado de *write-back*.
- A Figura 5 ilustra a execução do quarto ciclo de clock para as operações lógico-aritméticas.

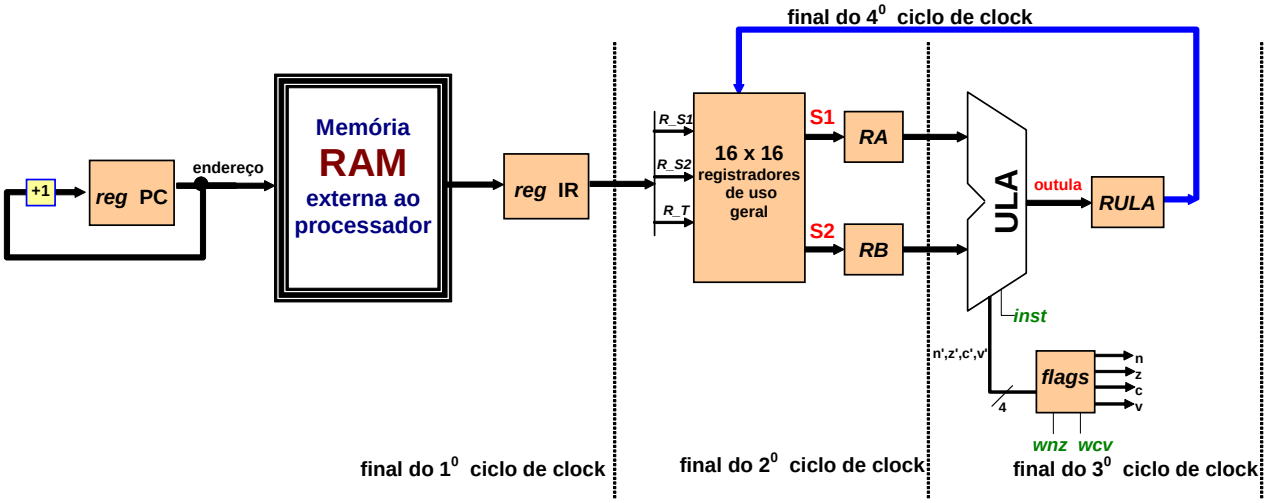


Figura 5 - Fluxo de execução de instruções lógico-aritméticas.

- A operações com modo de endereçamento imediato implicam em um dado registrador destino (*target*) receber o resultado de uma dada operação entre o próprio registrador *target* e uma constante de 8 bits. As operações em modo de endereçamento imediato são:
  - carga da parte alta de um registrador (LDH): *Rt ← constante & RtL* (o registrador *target* recebe a constante na parte alta e mantém a parte baixa inalterada).
  - carga da parte baixa de um registrador (LDL): *Rt ← RtH & constante* (o registrador *target* recebe a constante na parte baixa e mantém a parte alta inalterada).
  - soma/subtração em modo imediato: 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 para gerar um valor de 16 bits **positivo**.

Importante: para inicializar um registrador com uma constante de 16 bits devemos utilizar 2 instruções, LDH e LDL. Para ler/escrever um dado contido em um endereço (16 bits) são necessárias 3 instruções em linguagem de montagem: as duas primeiras carregam em um registrador a parte alta e baixa de um endereço (LDH e LDL, respectivamente) e a terceira instrução realiza a leitura/escrita (LD ou ST).

- Exemplo:

```
XOR   R0,R0,R0      ; zera o registrador R0
LDH   R1, #03H      ; #: indica valor imediato. H: indica que o valor está em hexadecimal.
LDL   R1, #27H      ; armazena no registrador R1 o valor 0327H
LD    R5, R1, R0    ; armazena em R5 o conteúdo do endereço armazenado em R1+R0
```

5.4.2 Execução da instrução de leitura de dados da memória (LD)

- Registrador *target* recebe o conteúdo da posição de memória endereçada pela soma dos conteúdos de dois registradores fonte (*sources*): *Rt ← PMEM(Rs1 + Rs2)*. Um dos registradores pode ser considerado como registrador base e o segundo como registrador contendo o deslocamento (*offset*).
- No quarto ciclo de clock 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 de leitura na memória é apresentada na Figura 6.

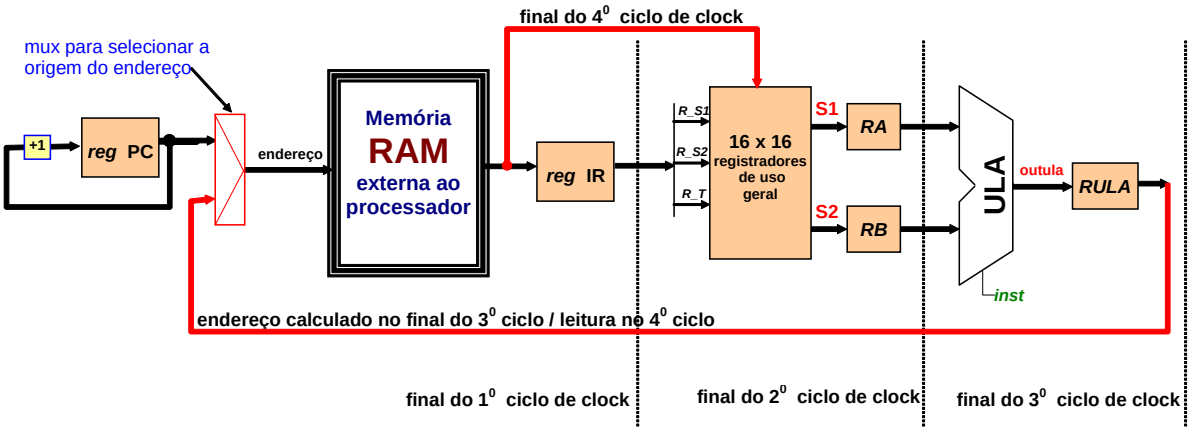


Figura 6 - Fluxo de execução de instrução de leitura na memória.

5.4.3 Execução da instrução de escrita em memória (ST)

- Posição de memória endereçada pela soma dos conteúdos de dois registradores fonte (*sources*) recebe o conteúdo do registrador destino (*target*): *PMEM(Rs1 + Rs2)* ← *Rt*.
- No quarto ciclo de clock o registrador endereçado por IR[11:8] é lido, gravando-se o conteúdo deste no endereço definido por *RULA*.
- Endereça a memória, gravando-se o conteúdo do registrador *target* (S2) na memória.
- Uma possível organização do bloco de dados para a execução da instrução de escrita na memória é apresentada na Figura 7.

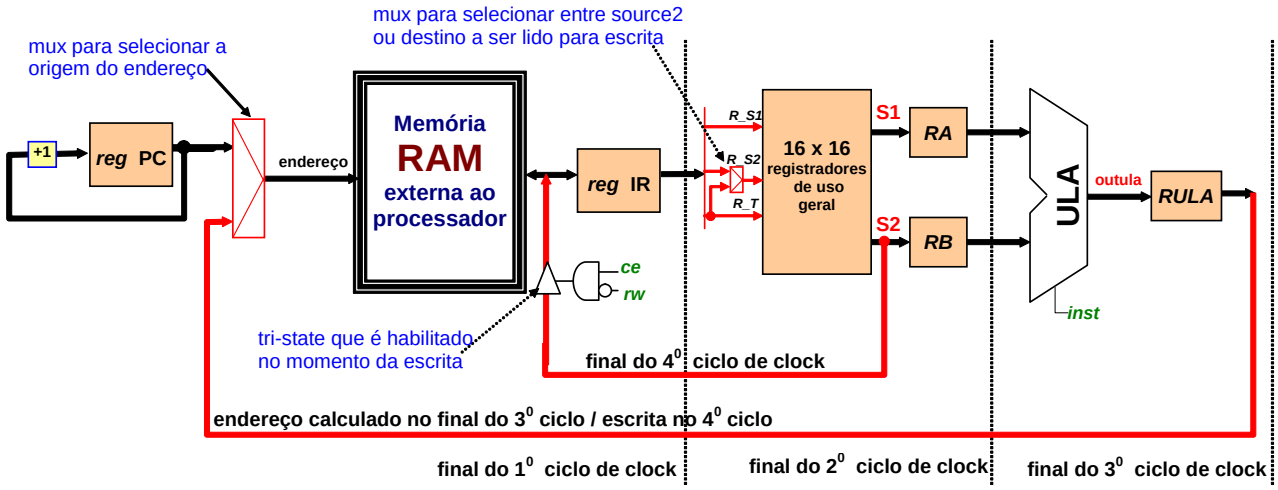


Figura 7 - Fluxo de execução de instrução de escrita na memória.

5.4.4 Operações de saltos incondicionais e condicionais

- O endereço destino do salto foi calculado no terceiro ciclo de clock, estando este armazenado no registrador *RULA*.
- Caso seja um salto condicional e o respectivo *flag* for igual a zero, a instrução é finalizada no terceiro ciclo (PC não é alterado).
- Caso o salto deva ser executado, o PC deve receber o conteúdo de *RULA*.
- Uma possível organização do bloco de dados para a execução dos saltos é apresentada na Figura 8.

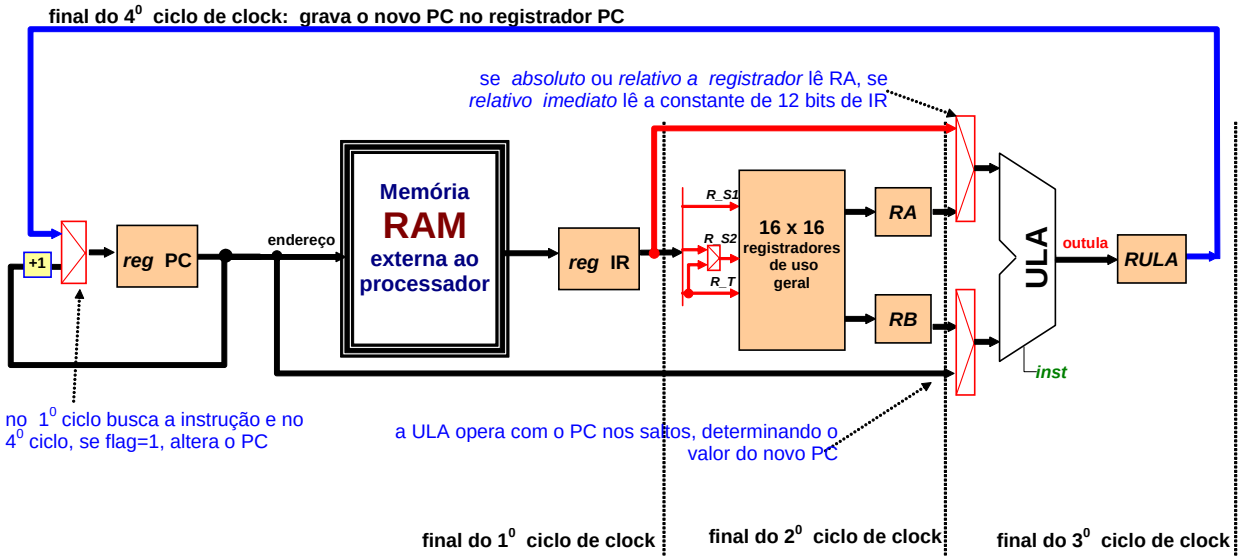


Figura 8 - Fluxo de execução de saltos incondicionais/condicionais.

5.4.5 Operações de chamada a subrotina

As operações que manipulam o registrador SP (*Stack Pointer*) são: inserção/remoção de registrador na pilha (PUSH/POP), chamadas a subrotinas (JSR, JSRR, JSR), inicialização do registrador SP (LDSP) e retorno de subrotina (RTS). A

Figura 9 ilustra o funcionamento da pilha.

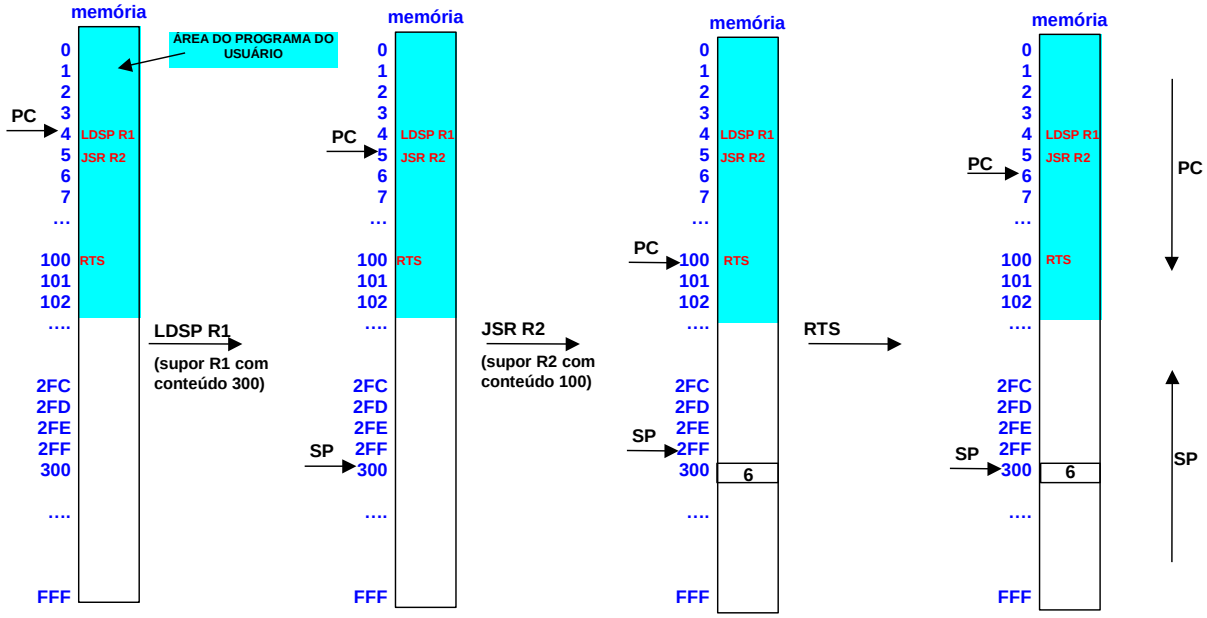


Figura 9 - Operação da pilha.

- O programa é armazenado do endereço *0* até um endereço *N*, logo, os endereços de programa crescem com os endereços da memória.
- A pilha cresce no sentido **inverso** da memória (*Patterson, ed. Bras. Página 71-72*). A razão para isto é não interferir com a área de dados e programa.
- O conteúdo do registrador SP **sempre aponta para a primeira posição livre da pilha**.

A execução da chamada a subrotina é feita da seguinte forma:

- O endereço destino do salto foi calculado no terceiro ciclo de clock, estando este armazenado no registrador *RULA*.
- Caso seja um salto condicional e o respectivo *flag* for igual a zero, a instrução é finalizada no terceiro ciclo (PC não é alterador).

- Caso o salto deva ser executado, o PC é armazenado no topo da pilha antes de receber o conteúdo de *RULA* e decrementa-se o SP:

PMEM(SP)←PC;
SP←SP-1;
PC← *resultado de RULA (PC+offset* ou *RS1* ou *PC+RS1*)

- Uma possível organização do bloco de dados para a execução dos saltos é apresentada na Figura 10.

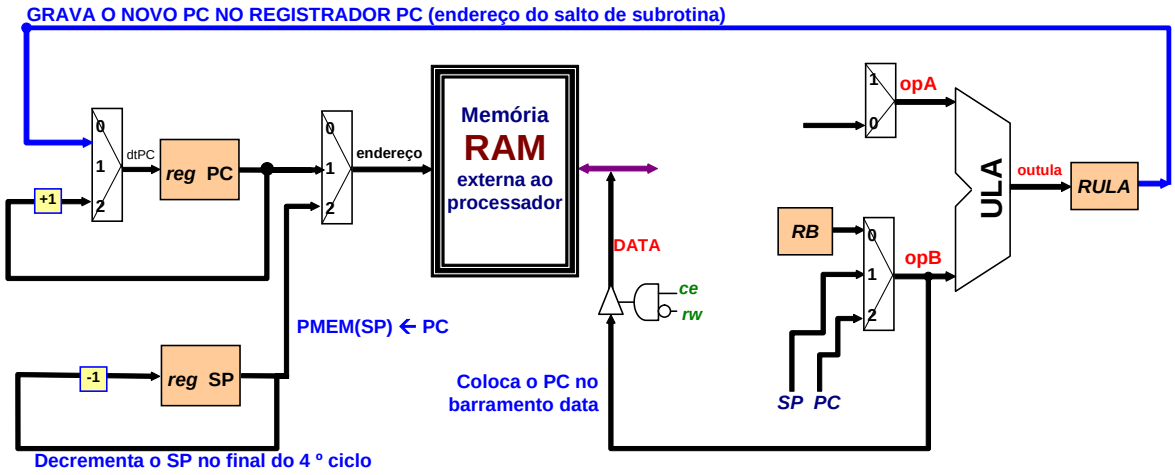


Figura 10 - Fluxo de execução para chamada a subrotina.

- A execução da instrução PUSH é semelhante à chamada de subrotina:

PMEM(SP)←Rt; (armazenado em *RB*)
SP←SP-1;

5.4.6 Operações de retorno de subrotina (RTS) e recuperação de registrador do topo da pilha (POP)

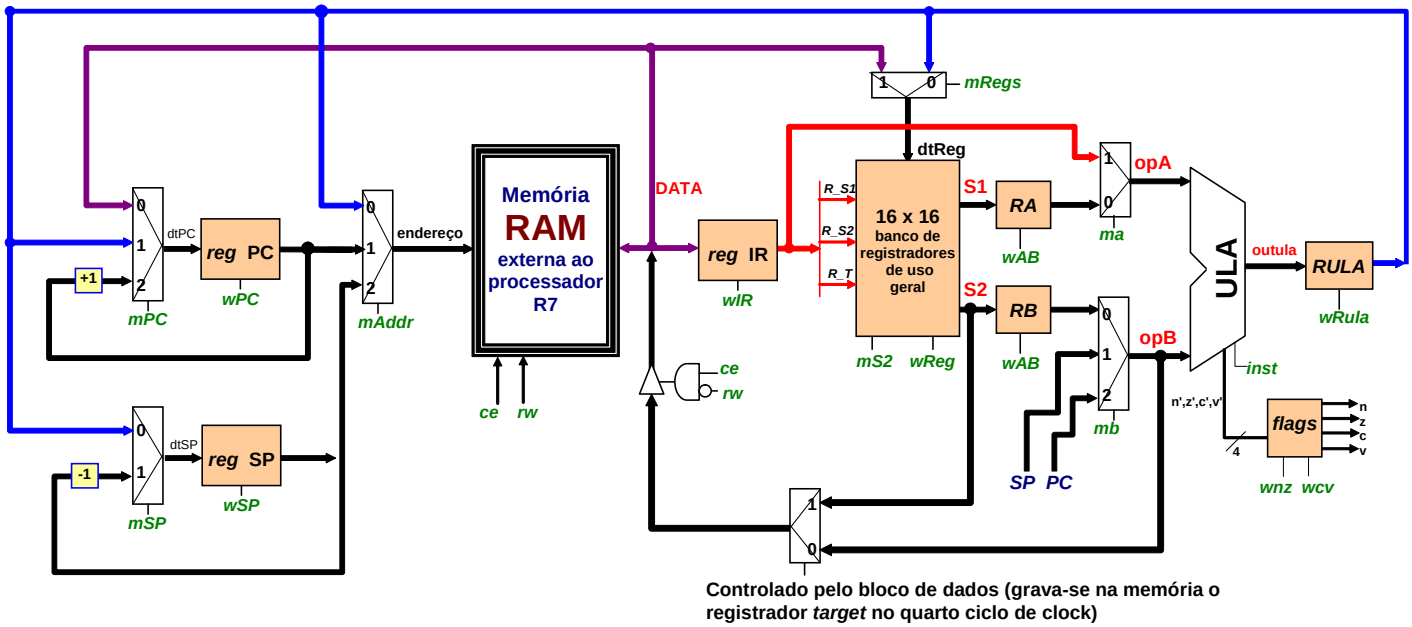
- O quarto ciclo das instruções RTS/POP implica endereçar a memória com o valor do registrador *RULA* (ciclo 3: RULA←SP+1), gravando o resultado da leitura ou no registrador PC (RTS) ou no registrador destino (POP). O registrador SP é atualizado com o valor do registrador *RULA* (SP+1).

6 ORGANIZAÇÃO DO BLOCO DE DADOS

Unindo as diversas figuras anteriores, obtemos o diagrama da Figura 11. Alguns elementos adicionais foram inseridos para o controle de subrotinas, devido à necessidade de manipular o registrador SP (*Stack Pointer*). O Bloco de Dados envia ao Bloco de Controle o conteúdo do registrador IR e o conteúdo dos *flags* de estado.

O bloco de dados necessita ***18*** sinais de controle, organizados em 4 classes:

- habilitação de escrita em registradores (8): ***wPC***, ***wSP***, ***wIR***, ***wAB***, ***wRula***, ***wRegs***, ***wnz***, ***wcv***.
- controle de leitura/escrita na memória externa (2): ***ce*** e ***rw***.
- controle de multiplexadores (7): ***mPC*** (origem dos dados para o PC), ***mSP*** (origem dos dados para o SP), ***mAddr*** (qual registrador endereça a memória), ***mRegs*** (origem dos dados para o banco de registradores), ***mS2*** (qual campo do IR seleciona o segundo operando), ***ma*** (origem dos dados para o primeiro operando da ULA), ***mb*** (origem dos dados para o segundo operando da ULA).
- a operação que a unidade lógica-aritmética executa (1): ***inst***.



Nesta figura estão representados todos os **18** 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.

Figura 11 - Bloco de dados completo (mais memória externa).

A Figura 12 ilustra a organização do banco de registradores, sob forma de um diagrama de blocos. Observar que o multiplexador de seleção do segundo registrador fonte (S2) está dentro do bloco "banco de registradores".

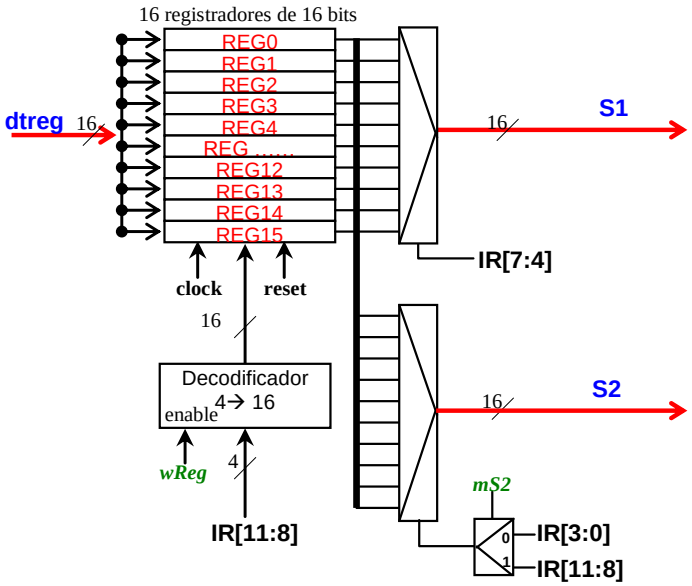
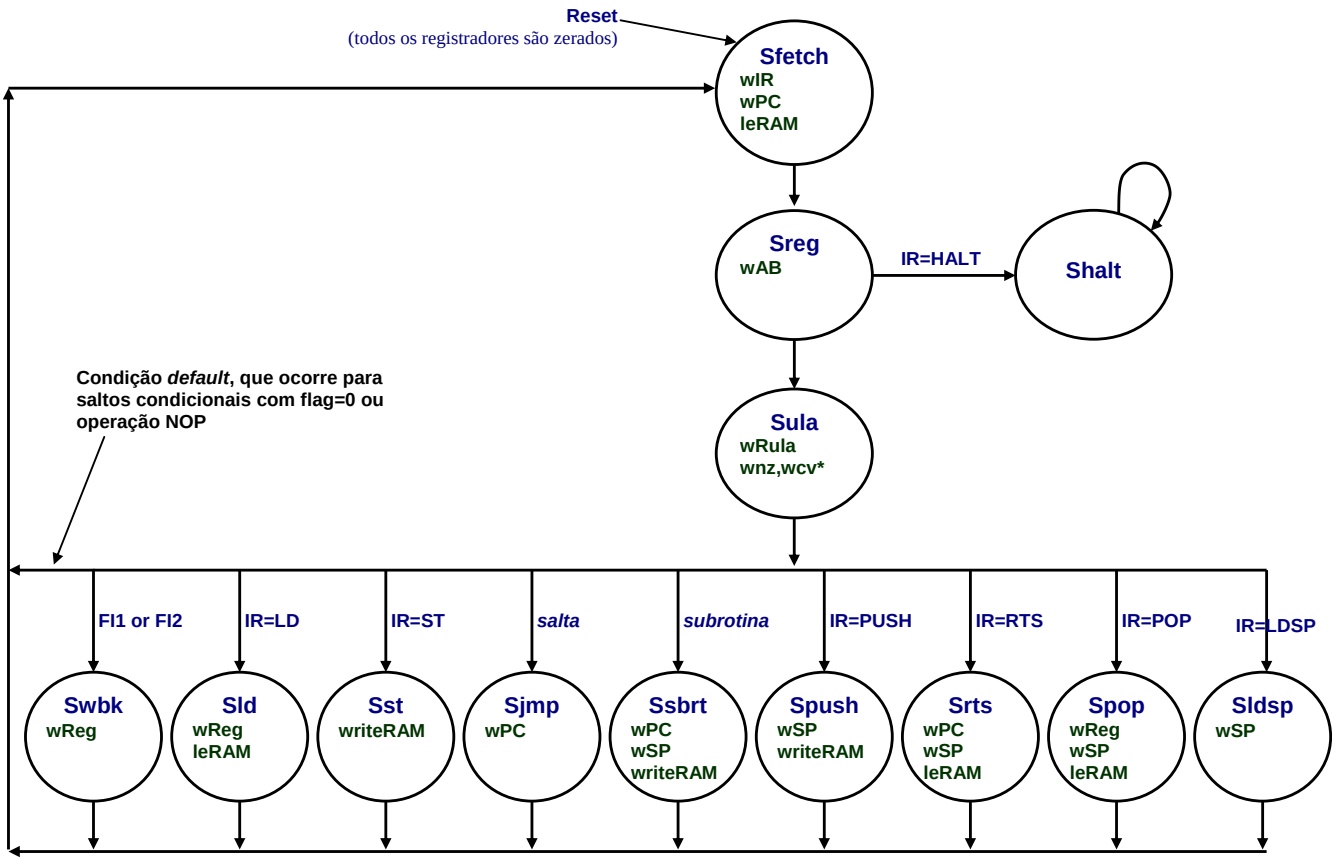


Figura 12 - Diagrama em blocos do banco de registradores de uso geral.

7 BLOCO DE CONTROLE

Pra executar qualquer instrução neste processador, é necessário definir uma máquina de estados. A Figura 13 ilustra esta máquina de estados, onde o próximo estado é função 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 *wrtieRAM*).



- FI1 – Formato de Instrução tipo 1 – Registrador *target* **não** é fonte (Ex: ADD R1, R2, R3)
- FI2 - Formato de Instrução tipo 2 - Registrador *target* **também é** fonte (Ex: ADDI R1, #2)
- *Salta*: avaliação das 15 instruções de salto
- *Subrotina*: JSR, JSRR, JSRD
- * - a escrita nos *flags* no estado *Sula* depende do tipo de operação

Figura 13 - Máquina de estados de controle.

A função dos 13 diferentes estados é:

- Sfetch: primeiro ciclo de clock, busca a instrução;
- Srreg: segundo ciclo de clock, decodificação da instrução e leitura dos registradores fontes;
- Shalt: segundo ciclo de clock, finaliza a execução e aguarda reset;
- Sula: terceiro ciclo de clock, operação com a ULA;
- Swbk: quarto ciclo de clock, armazena resultado da operação da ULA no registrador destino;
- Sld: quarto ciclo de clock, busca dado da memória e armazena no registrador destino;
- Sst: quarto ciclo de clock, salva registrador destino na memória;
- Sjmp: quarto ciclo de clock, altera o PC em saltos incondicionais ou condicionais com *flag=1*;
- Ssbrt: quarto ciclo de clock, salta para subrotina;
- Spush: quarto ciclo de clock, coloca registrador no topo da pilha;
- Srts: quarto ciclo de clock, retorna de subrotina;
- Spop: quarto ciclo de clock, retira registrador do topo da pilha;
- Sldsp: quarto ciclo de clock, inicializa o registrador SP (topo da pilha);

7.1 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 no *package* da descrição do processador um tipo enumerado contendo todas as possíveis instruções do processador. Originalmente o processador contém **40** instruções, mas agrupando-se os saltos em grupos (saltoR, saltoA, saltoD), fica-se em **28** instruções.

```
type instrucao is
    ( add,  sub,  e,   ou,   oux,  addi,  subi,  ldl,   ldh,   ld,  st,  sl0, sl1, sr0,  sr1,
      notA, nop, halt, ldsp, rts,  pop,   push,  saltoR, salto, saltoD, jsrr,  jsr, jsrd);
```

- Decodificação das instruções. Observar que para efeito de simplificação do programa os 5 saltos relativos foram agrupados em "saltaR", os 5 saltos absolutos em "saltoA" e os 5 saltos com deslocamento curto em "saltoD". Esta simplificação reduz o código VHDL que será escrito.

```
i <= add   when ir(15 downto 12)=0  else
     sub   when ir(15 downto 12)=1  else
     ……
     sr1   when ir(15 downto 12)=11 and ir(3 downto 0)=3  else
     ……
     saltoR when ir(15 downto 12)=12 and  ( ir(3 downto 0)=0 or
                 (ir(3 downto 0)=1 and fn='1') or (ir(3 downto 0)=2 and fz='1') or
                 (ir(3 downto 0)=3 and fc='1') or (ir(3 downto 0)=4 and fv='1') )   else
     saltoA  when ……
     saltoD  when …
     …
     jsr   when ir(15 downto 12)=12 and ir(3 downto 0)=11 else
     jsrd  when ir(15 downto 12)=15;

uins.inst <= i;       --- ************ operacao que a ula ira' executar
```

- Decodificação das instruções com formato tipo 1:

```
instructionFormat1 <= '1' when i=add or i=sub or i=e or i=ou or i=oux or i=notA or i=sl0 or
                              i=sr0 or i=sl1 or i=sr1 else
                      '0';
```

- Decodificação das instruções lógico-aritméticas do tipo 2 (Rt no lado direito da expressão):

```
instructionFormat2 <= '1' when i=addi or i=subi or i=ldl or i=ldh else
                      '0';
```

7.2 Controle dos multiplexadores

Os sinais de controle dos multiplexadores (7) dependem do estado atual da máquina de controle (EA) e/ou da instrução corrente. Recomenda-se localizar os controles dos multiplexadores na Figura 11. Os sinais de controle são precedidos do sufixo "*uins.*", que designa a *record* microinstrução.

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

```
uins.mPC <= "10" when EA=Sfetch else     -- na  busca incrementa o PC
            "00" when EA=Srts  else      -- em  retorno de subrotina busca da memória
            "01";                        -- por default traz da ULA (retorno de subrotina)
```

2. Controle da origem dos dados para o SP (depende apenas da instrução):

```
uins.mSP <= '1' when  i=jsrr or i=jsr or i=jsrd or i=push else  -- pos-decremento
            '0';
```

3. Controle da origem dos dados para o endereçamento da memória (depende do estado de controle):

```
uins.mAddr <= "10" when EA=Spush or  EA=Ssbrt else  -- em subrotina o SP endereça a memória
              "01" when EA=Sfetch else              -- na  busca o PC endereça a memória
              "00";                                 -- por default: LD/ST
```

4. Controle da origem dos dados para o escrita nos registradores (depende apenas da instrução):

```
uins.mRegs <= '1' when i=ld or I=pop else '0';     -- escreve nos registradores o conteúdo vindo
                                                   -- da memória
```

5. Escolha do segundo operando (depende da instrução e do estado atual). O segundo fonte (*source2*) recebe o endereço do registrador destino quando for uma instrução com formato tipo 2 ou operação de escrita na memória .

```
uins.mS2 <= '1' when instructionFormat2='1' or i=push or EA=Sst else '0';
```

6. Controle da origem dos dados para a ULA (depende apenas da instrução):

```
-- primeiro operando da ULA é o IR quando instrução com o formato do tipo 2 ou jump/jsr com
-- deslocamento curto
 uins.ma <= '1' when instructionFormat2='1' or i=saltoD or i=jsrd else '0';


-- segundo multiplexador
 uins.mb <= "01"  when  i=rts or i=pop   else      -- para incrementar o SP
            "10"  when  i=saltoR or i=saltoA or i=saltoD or i=jsrr or i=jsr or i=jsrd  else
            "00" ;
```

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

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

8 EXECUÇÃO DE UMA SEQÜÊNCIA DE OPERAÇÕES

A simulação da Figura 14 ilustrada a execução das 5 últimas instruções do trecho de código abaixo:

```
end    instrução
0128   7190
0129   8101;  R1 ← 0190          (400 em decimal)
012A   73AA
012B   83BB;  R3 ← BBAA
012C   AD01;  grava o conteúdo do registrador D no endereço contido no registrador 1 (190H ou 400)
012D   9F10;  lê o conteúdo do endereço contido no registrador 1, gravando no registrador 15
```

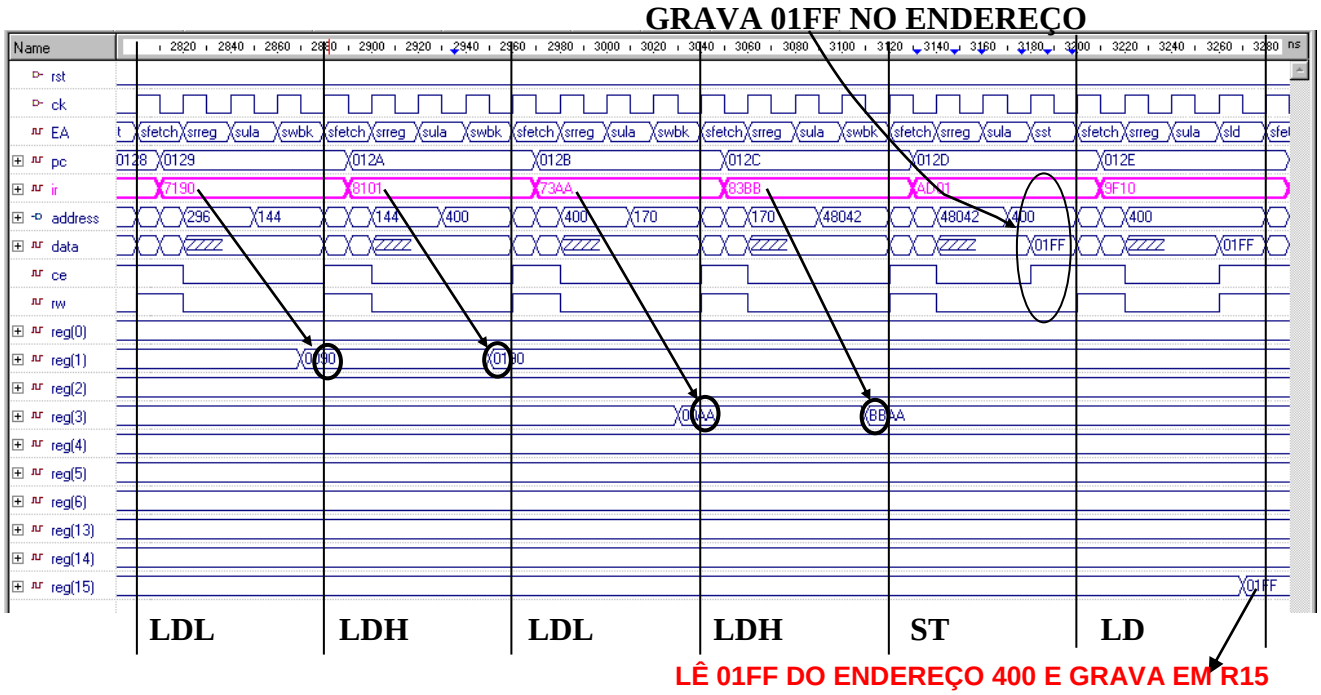


Figura 14 - Simulação de 6 instruções.

9 PROGRAMA EXEMPLO PARA TESTAR TODAS AS INSTRUÇÕES

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 no formato "xxxx yyyy", onde xxxx é o endereço com 16 bits (4 dígitos hexadecimais) e yyyy é a instrução com 16 bits (4 dígitos hexadecimais). O arquivo de teste é carregado na memória quando o reset é ativado, no início da simulação.

```
0000 7108    ;
0001 8110    ;  LOAD R1, #1008
0002 7234    ;
0003 8212    ;  LOAD R2, #1234
0004 73DC    ;
0005 83FE    ;  LOAD R3, #FEDC
0006 0412    ; soma:      resultado em R4: 223C
0007 1512    ; subtrai:   resultado em R5: FDD4
0008 2612    ; and:       resultado em R6: 1000
0009 3712    ; or:        resultado em R7: 123C
000A 4812    ; xor:       resultado em R8: 023C
000B 5101    ; soma imedidato 01 no R1   - 1009
000C 510F    ; soma imedidato 0F no R1     - 1018
000D 51FF    ; soma imedidato FF no R1   - 1117  (1130 ns)
000E 6201    ; sub. imedidato 01 de R2   - 1233
000F 6204    ; sub. imedidato 04 de R2   - 122F
0010 62FF    ; sub. imedidato FF de R2   - 1130
0011 7DFF
0012 8D01    ;  LOAD RD, #01FF
0013 B0D7    ;  carrega o topo da pilha com o conteúdo do registrador RD    (511 em decimal)  (1610 ns)
0014 7100
0015 8101    ;  R1 <- 0100
0016 C01B    ; ********* salta para subrotina apontada pelo R1 (endereco 0100)  *********
0017 B005    ; nop
0018 70FF
0019 80FF    ;  R0 <- FFFF      /  seta flag negativo
001A 50FF    ;  R0 <- R0 + FF   / seta o flag de overflow
001B 4000    ;  RO <- R0 xor RO / seta o flag zero
001C 7730    ;
001D 8700    ;  R7 <- 0030
001E C077    ;  salta para o endereco apontado por R7 (30H) se flag z setado

0030 7710
0031 8700
0032 C070    ; salto incondicional relativo para o endereco 33H+10H=43H

0043 D050    ; salto para o endereco 44H+50H=94H

0094 B006    ; ********** HALT HALT *******  4800 NS DE SIMULACAO  ****************

0100 B10A    ; SUBROTINA QUE TESTA O EMPILHAMENTO E DESEMPILHAMENTO DE REGISTRADORES
0101 B20A
0102 B30A
0103 B40A    ; empilha os registradores 1 a 4
0104 7109
0105 8100
0106 C01A    ; aqui CHAMA OUTRA SUBROTINA -MODO RELATIVO A PC (110H-107H=09H)
0107 B409
0108 B309
0109 B209
010A B109    ; recupera da pilha os registradores 1 a 4 da pilha
010B B008    ;  ****** rts *****************      (4000 ns de simulacao)

0110 4111    ; SUBROTINA QUE zera com xor os registradores 1 a 4 utilizando XOR - 2490 ns
0111 4222
0112 4333
0113 4444    ; tempo de simulacao: 2730 ns
0114 F013    ;  subrotina relativo ao PC, sala 13 palavras indo para o endereço 0128
0115 B008    ;  ****** rts *****************

0128 7190    ; SUBROTINA QUE TESTA O LOAD E O STORE
0129 8101    ; r1 <- 0190      (400 em decimal)
012A 73AA
012B 83BB    ; R3 <- BBAA
012C AD01    ; grava o conteúdo do registrador D no endereço contido no registrado 1 (190H ou 400)
012D 9F10    ; lê o conteúdo do endereço contido no reg 1, gravando no reg 15     (3250 ns)
012E B230
012F B220
0130 B221
0131 B221    ; testa sl0 e sl1   -  R2 resulta em BAA3
0132 B422
0133 B442
0134 B443
0135 B443    ; testa sr0 e sr1   - R4 resulta em CBAA - 4000 ns de simulação
0136 B104    ; not     - R1 resulta em FFFF
0137 B008    ;  ****** rts *****************
```

Recomenda-se **escrever os programas em linguagem de montagem** (*assembly*), gerando-se o código objeto automaticamente, a partir do montador/simulador.

A Figura 15 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.

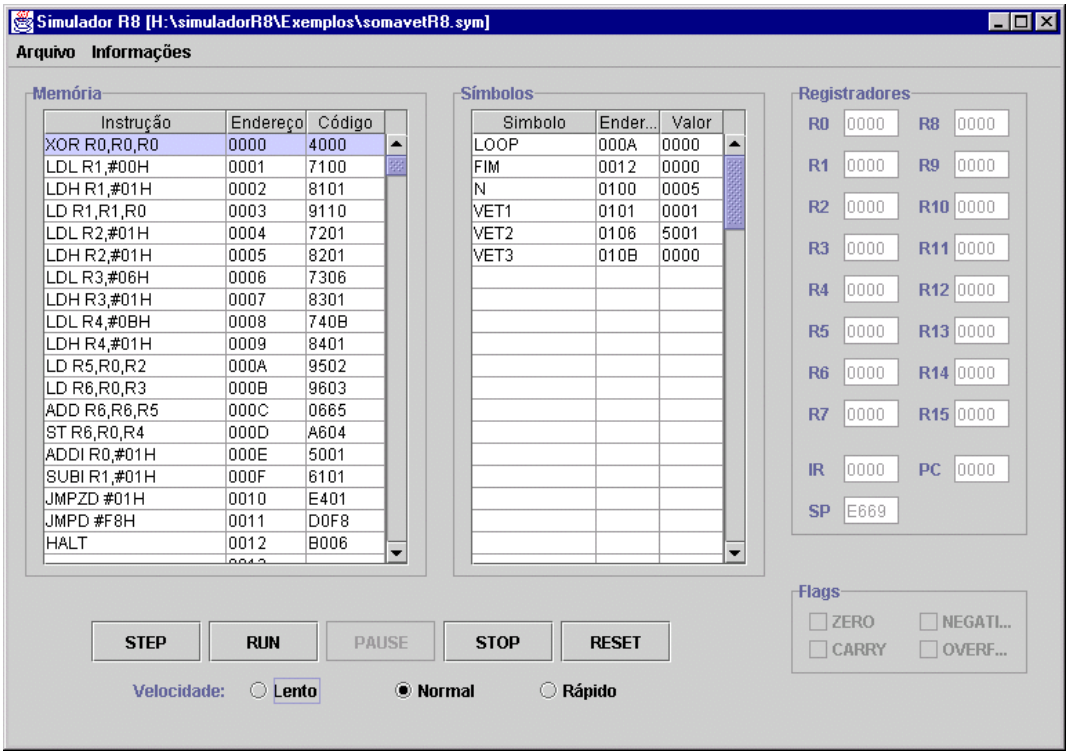


Figura 15 - Simulador R8.

A ferramenta de montagem tem como entrada o nome do programa em linguagem descrito em linguagem *assembly* (<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 VHDL**.

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 *assembly* não condizem com as instruções existentes na arquitetura.