Última alteração: 25/Abril/2001 por Fernando Moraes

PROCESSADOR MULTI-CICLO

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 acesso 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*, assunto que será visto ao final da disciplina.

Características específicas do nosso processador multi-ciclo:

- dados e endereços são de 16 bits (processador de 16 bits);
- endereçamento de memória a palavra (cada endereço corresponde a 16 bits de conteúdo);
- banco de registradores com 16 registradores de uso geral;
- apenas 1 *flag* de estado;
- todos os saltos e chamadas de subrotina são condicionais ao *flag* de estado. Não existem saltos incondicionais (restrição imposta para simplificar o conjunto de instruções). É possível simular saltos condicionais com pseudo-instruções a nível de linguagem de montagem;
- execução das instruções em 3 ou 4 ciclos, ou seja, CPI entre 3 e 4.

2 CONJUNTO DE INSTRUÇÕES

- Operações lógicas e aritméticas binárias: soma, subtração, E, OU, OU exclusivo.
- Operações lógicas e aritméticas com constantes curtas: soma, subtração.
- Operações unárias: deslocamento para direita ou esquerda e inversão (NOT).
- Carga de metade de um registrador com uma constante (LDL e LDH).
- Inicialização do *flag* de estado por 4 instruções específicas: DIF, EQL, SUP, INF.
- Inicialização do apontador de pilha (LDSP) e retorno de subrotina (RTS).
- NOP (no operation): operação vazia (útil para laços de espera e reserva para código futuro).
- 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)

- Lembrando:
  - Todos os saltos são condicionais
  - As comparações de maior ou menor são para números inteiros POSITIVOS

Convenções utilizadas:

RtH: **oito bits mais significativos de Rt**
RtL: **oito 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 = Rtarget [destino], Rs1 = Rsource1, Rs2 = Rsource2

Instrução	FORMATO DA INSTRUÇÃO				AÇÃO
	15 – 12	11 - 8	7 - 4	3 - 0	
ADD Rt, Rs1,Rs2	**0**	R target	R source1	R source2	Rt ← Rs1 + Rs2
SUB Rt, Rs1,Rs2	**1**	R target	R source1	R source2	Rt ← Rs1 - Rs2
AND Rt, Rs1,Rs2	**2**	R target	R source1	R source2	Rt ← Rs1 *and* Rs2
OR Rt, Rs1,Rs2	**3**	R target	R source1	R source2	Rt ← Rs1 *or* Rs2
XOR Rt, Rs1,Rs2	**4**	R target	R source1	R source2	Rt ← Rs1 *xor* Rs2
ADDI Rt, *cte8*	**5**	R target	Constante		Rt ← Rt + ("00000000" & constante)
SUBI Rt, *cte8*	**6**	R target	Constante		Rt ← Rt - ("00000000" & constante)
LDL Rt, *cte8*	**7**	R target	Constante		Rt ← RtH & constante
LDH Rt, *cte8*	**8**	R target	Constante		Rt ← constante & RtL
LD Rt, Rs1,Rs2	**A**	R target	R source1	R source2	Rt ← PMEM (Rs1+Rs2)
ST Rt, Rs1,Rs2	**B**	R target	R source1	R source2	PMEM (Rs1+Rs2) ← Rt
SL Rt, Rs1	**C**	R target	**0**	R source2	Rt ← Rs2[14:0] & *flag*
SR Rt, Rs1	**C**	R target	**1**	R source2	Rt ← *flag* & Rs2 [15:1]
NOT Rt, Rs1	**C**	R target	**2**	R source2	Rt ← not (Rs2)
DIF Rs1, Rs2	**D**	**0**	R source1	R source2	if *Rs1* ≠ *Rs2* then *flag=1* else *flag=0*
EQL Rs1, Rs2	**D**	**1**	R source1	R source2	if *Rs1* = *Rs2* then *flag=1* else *flag=0*
SUP Rs1, Rs2	**D**	**2**	R source1	R source2	if *Rs1* > *Rs2* then *flag=1* else *flag=0*
INF Rs1, Rs2	**D**	**3**	R source1	R source2	if *Rs1* < *Rs2* then *flag=1* else *flag=0*
LDSP Rs1	**D**	**4**	R source1	-	SP ← Rs1 (inicializa o apontador de pilha)
NOP	**D**	**5**	-	-	nenhuma ação
RTS	**D**	**6**	-	-	PC ←PMEM(SP+1) ; SP←SP+1
HALT	**D**	**7**	-	-	suspende seqüência de ciclos de busca e execução
JMPR Rs1	**D**	**8**	R source1	-	if (*flag* =1) PC ← PC + Rs1

JSRR Rs1	**D**	**9**	R source1	-	if (*flag* =1) { PMEM(SP)←PC ;
					SP←SP-1;
					PC← PC+ Rs1
					}

JMP Rs1	**D**	**A**	R source1	-	if (*flag* =1) PC ← Rs1

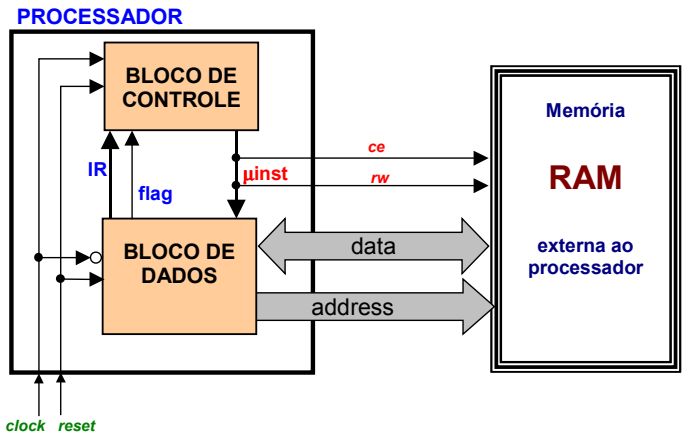
JSR Rs1	**D**	**B**	R source1	-	if (*flag* =1) { PMEM(SP)←PC ;
					SP←SP-1;
					PC← Rs1
					}

JMPD desloc	**E**	Deslocamento			if (*flag* =1) PC ← PC + **ext_sinal &** desloc

JSRD desloc	**F**	Deslocamento			if (*flag* =1) { PMEM(SP)←PC ;
					SP←SP-1;
					PC← PC + **ext_sinal &** desloc
					}

3 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.



(a) Diagrama de blocos processador-memória

PROCESSADOR

```
entity processador is
      port( ck,rst: in std_logic;
            data: inout reg16;
            address: out reg16;
            ce,rw: out std_logic );
end processador;

architecture processador of processador is
      signal flag: std_logic;
      signal uins: microinstrucao;
      signal ir: reg16;
   begin

      dp: datapath
          port map( uins=>uins, ck=>ck, rst=>rst,
                    instrucao=>ir,endereco=>address,
                    data=>data, flag=>flag);


      ctrl: controle
            port map(uins=>uins, ck=>ck, rst=>rst,
                     flag=>flag, ir=>ir);
      ce <= uins.ce;
      rw <= uins.rw;

end processador;
```

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

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

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

O bloco de dados envia para o bloco de controle a instrução corrente (***IR***) e o qualificador de estado (***flag***). 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* (barramento bidirecional de 16 bits para os dados) e *address* (barramento de 16 bits com os endereços de memória).

O controle de acesso à memória é feito pelo bloco de controle, através dos sinais *ce* e *rw*. O sinal *ce* indica operação com a memória e o sinal *rw* indica operação ou de escrita ou de leitura.

É importante ressaltar que os blocos de dados e controle operam em fases distintas do sinal *clock*. Em uma borda do *clock* (por exemplo, subida) o bloco de controle gera a micro-instrução, e na borda seguinte (descida) o bloco de dados modifica os registradores. Com isto sempre teremos dados estáveis nas transições de clock.

A Figura 1b representa o 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*.

4 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*): contador de programa.
- SP (*stack pointer*): 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).

Além destes registradores há um *bit* de estado, denominado ***flag***, utilizado para controle dos saltos e chamadas a subrotinas. O estado do *flag*, 0 ou 1, é determinado por instruções de comparação entre registradores (igualdade, diferença, maior e menor).

Registradores temporários são utilizados entre os estágios de execução. 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*.

5 EXECUÇÃO DAS INSTRUÇÕES NO BLOCO DE DADOS

5.1 Instruções lógica-aritméticas

- Registrador destino (*target*) recebe o resultado de uma dada operação binária entre registradores fonte (*sources*): Rt ← Rs1 op Rs2
- Formato da instrução:

Bits 15 - 12	Bits 11 - 8	bits 7 - 4	bits 3 - 0
Opcode	R target	R source1 ou opcode2	R source2

No nosso processador, o *opcode* indica 5 operações lógico-aritméticas entre 2 registradores fontes: soma, subtração, e lógico, ou lógico, ou exclusivo. Quando tivermos operações sobre apenas um registrador fonte, os bits 7 a 4 indicam o tipo de operação e os bits 15 a 12 permanecem constantes. As 3 operações assumidas por *opcode2* são: sl (deslocamento à esquerda), sr (deslocamento à direita) e not (inversão lógica).

- A execução de instruções lógico-aritméticas é realizada em 4 ciclos de relógio:
  1) *busca:* busca a instrução endereçada pelo registrador *PC* na memória, grava a instrução no registrador *IR* e incrementa o *PC*;

     IR ← PMEM(PC); PC++;

  2) *registrador:* lê os registradores fonte, endereçados implicitamente pela instrução contida no *IR*, armazenando-os nos registradores *RA* e *RB*;
  3) *ula:* a ULA realiza a operação especificada pela instrução armazenada no IR, armazenando o resultado no registrador *RULA*;
  4) *write-back:* grava o conteúdo do registrador *RULA* no registrador destino.

- A Figura 2 ilustra uma possível organização do bloco de dados para a execução das instruções lógico-aritméticas em 4 ciclos de relógio.

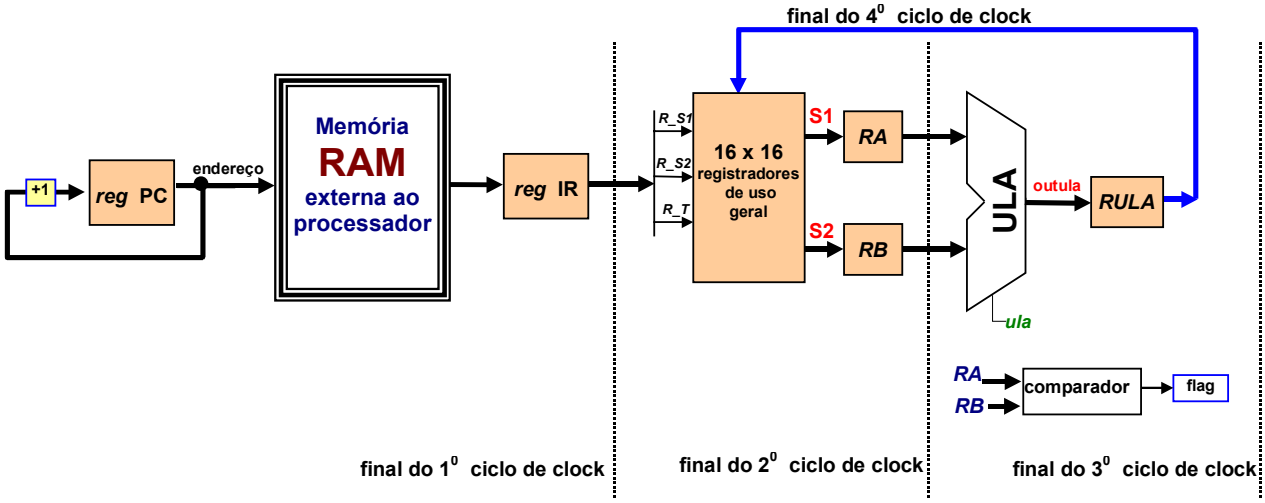


Figura 2 - Fluxo de execução de instruções lógico-aritméticas e comparações.

5.2 Instruções de comparação para atribuir valor ao *flag* de estado

- Registrador *flag* recebe o resultado de uma dada comparação entre dois registradores fonte (*sources*): *flag* ← Rs1 *comp* Rs2
- Formato da instrução:

Bits 15 - 12	Bits 11 - 8	bits 7 - 4	bits 3 - 0
Opcode	Opcode2	R source1	R source2

Opcode indica comparação e *opcode2* indica qual comparação: diferença (DIF), igualdade (EQL), maior que (SUP) e menor que (INF).

- As comparações são executadas em 3 ciclos de relógio. Ciclos para execução das instruções de comparação:
  1) *busca* (como no item instruções lógica-aritméticas)
  2) *registrador* (como no item instruções lógica-aritméticas)
  3) *comp:* compara os valores dos registradores *RA* e *RB*, armazenando o resultado no *flag* de estado.

5.3 Instrução de leitura de dados da memória (LOAD)

- Registrador *destino* 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*).
- Formato da instrução:

Bits 15 - 12	Bits 11 - 8	bits 7 - 4	bits 3 - 0
Opcode	R target	R source1	R source2

- Ciclos para leitura de uma posição da memória:
  1) *busca* (como no item instruções lógica-aritméticas)
  2) *registrador* (como no item instruções lógica-aritméticas)

3) *ula:* soma o conteúdo dos registradores fonte e determina o endereço de memória a ser lido.
4) *load:* grava o conteúdo lido da memória no registrador destino.

- Uma possível organização do bloco de dados para a execução da instrução de leitura na memória é apresentada na Figura 3.

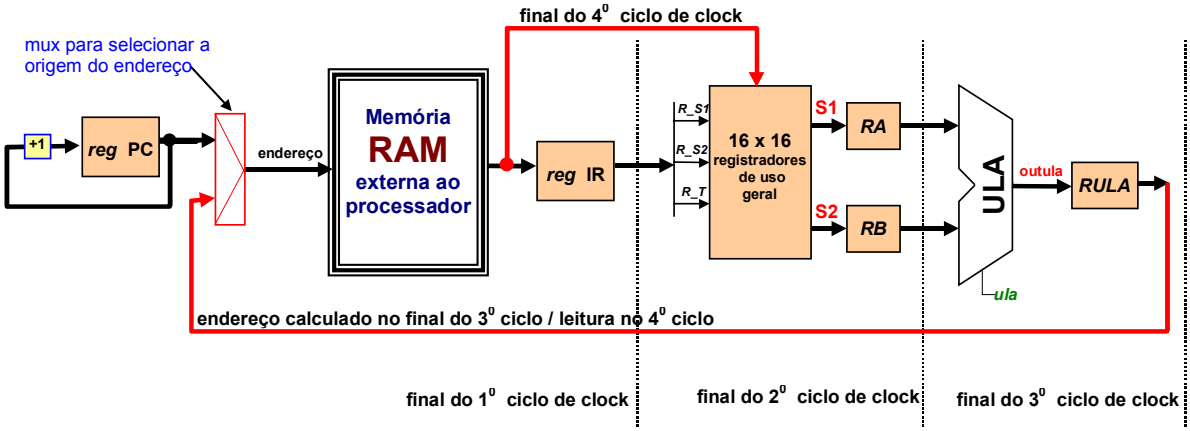


Figura 3 - Fluxo de execução de instrução de leitura na memória.

5.4 Instrução de escrita em memória (STORE)

- Posição de memória endereçada pela soma dos conteúdos de dois registradores fonte (*sources*) recebe o conteúdo do registrador destino (*target*): *PMEM(Rs1 + Rs2)* ← *Rt*.
- Mesmo formato da instrução *load*.
- Ciclos para escrita em uma posição da memória:
    1) *busca* (como no item instruções lógica-aritméticas)
    2) *registrador* (como no item instruções lógica-aritméticas)
    3) *ula:* soma o conteúdo dos registradores fonte e determina o endereço de memória a ser escrito.
    4) *store:* grava o conteúdo do registrador destino na memória. Observar que o registrador destino deve neste momento ser lido do banco de registradores, o que implica na inserção de um multiplexador, afim de selecionar entre leitura de endereço ou leitura do dado a ser escrito.

- Bloco de dados para a execução da instrução de escrita na memória (Figura 4):

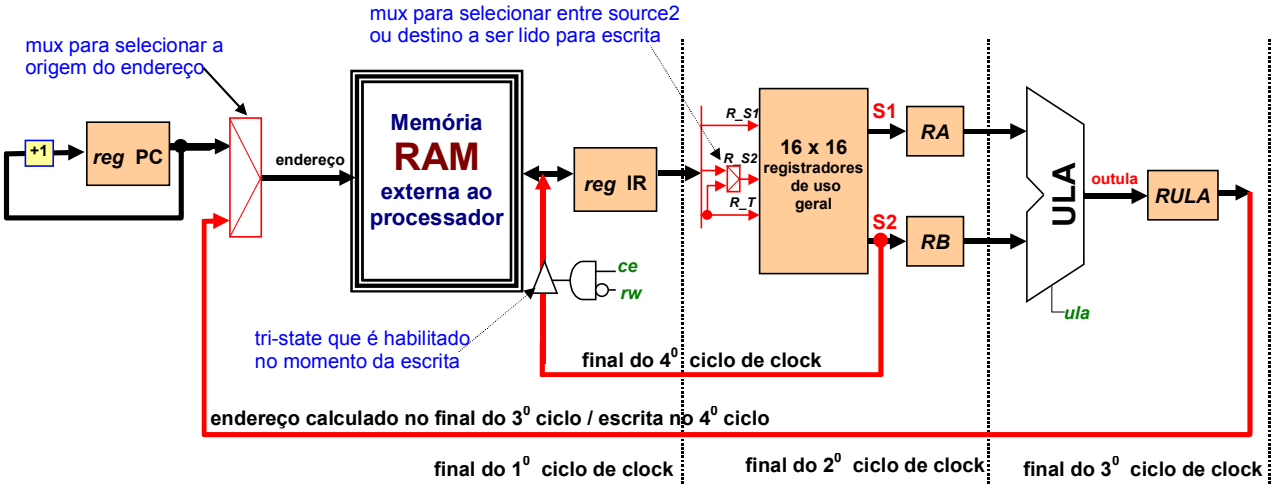


Figura 4 - Fluxo de execução de instrução de escrita na memória.

5.5 Operações em modo de endereçamento imediato

- Registrador destino (*target*) recebe o resultado de uma dada operação entre o próprio registrador *target* e uma constante de 8 bits (como todas as instruções são de uma palavra, o tamanho da constante não pode ser do tamanho da palavra).
- Formato da instrução:

Bits 15 - 12	Bits 11 - 8	bits 7 - 0
Opcode	R target	constante

- Operações:
  - 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*
  - soma/subtração em modo imediato: soma/subtração do conteúdo de um dado registrador a uma constante de 8 bits: *Rt* ← *Rt +/- constante*. Desta forma, podemos somar (ou subtrair) uma constante de valor 0 a 255 ao conteúdo de um dado registrador. **Importante**: a execução da instrução implica completar com zeros os 8 bits mais significativos da constante para gerar um valor de 16 bits.

- Ciclos para execução de instruções em modo imediato:
  1) *busca* (como no item instruções lógica-aritméticas)
  2) *registrador* (como no item instruções lógica-aritméticas)
  3) *ula:* realiza as operações de concatenação, soma e subtração.
  4) *write-back:* grava o conteúdo do registrador destino no banco de registradores.

- Bloco de dados para a execução de instruções em modo imediato (Figura 5):

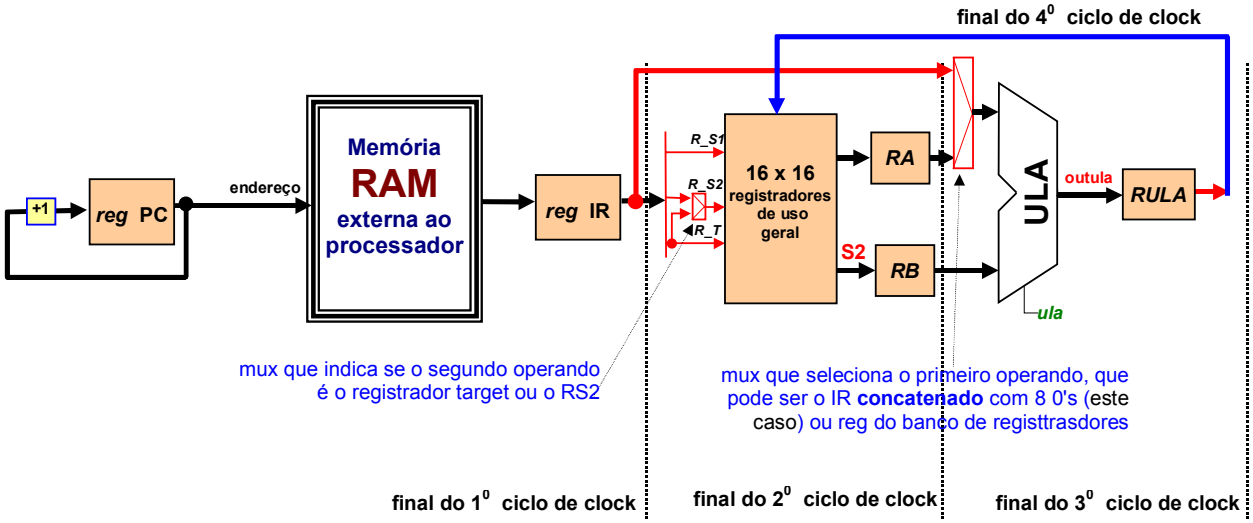


Figura 5 - Fluxo de execução de instruções em modo imediato.

Importante: para inicializar um registrador com uma constante de 16 bits devemos utilizar 2 instruções, LDL e LDH. 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
LDL   R1, #27H      ; armazena no registrador R1 o valor 0327H
LD    R5, R1, R0    ; armazena em R5 o conteúdo do endereço
                    ; obtido pela soma dos conteúdos de R1+R0
```

5.6 Operações de salto condicionais

- Numa operação de salto condicional, se o *flag* de estado estiver em '1', o PC é alterado conforme os operandos especificados na instrução. Há três formatos para o salto condicional:
  - registrador absoluto, JMP, quando o conteúdo do novo PC está em um registrador (PC ← RS1);
  - relativo a registrador, JMPR, quando um dado registrador contém um deslocamento em relação ao PC (PC ← PC + RS1);
  - relativo imediato, JMPD, o PC é somado a um deslocamento (PC ← PC + deslocamento).

- Formatos da instrução:

Bits 15 - 12	Bits 11 - 8	bits 7 - 4	bits 3 - 0
Opcode	Opcode2	R source1	-
Opcode	Deslocamento (constante)		

- Ciclos para execução de instruções em saltos condicionais:
  1) *busca* (como no item instruções lógica-aritméticas)
  2) *registrador* (como no item instruções lógica-aritméticas)
  3) *ula:* calcula o resultado do novo PC (quando for com deslocamento, a ULA deve realizar a extensão de sinal). Se flag=0 termina a instrução em 3 ciclos. *Uma possível otimização do processador é concluir a instrução em 1 ciclo caso flag=0. Isto é possível, pois o flag é calculado antes da execução da instrução, por instrução de comparação.*
  4) *altera_pc:* grava novo valor no registrador PC.

- Bloco de dados para a execução dos saltos condicionais (Figura 6):

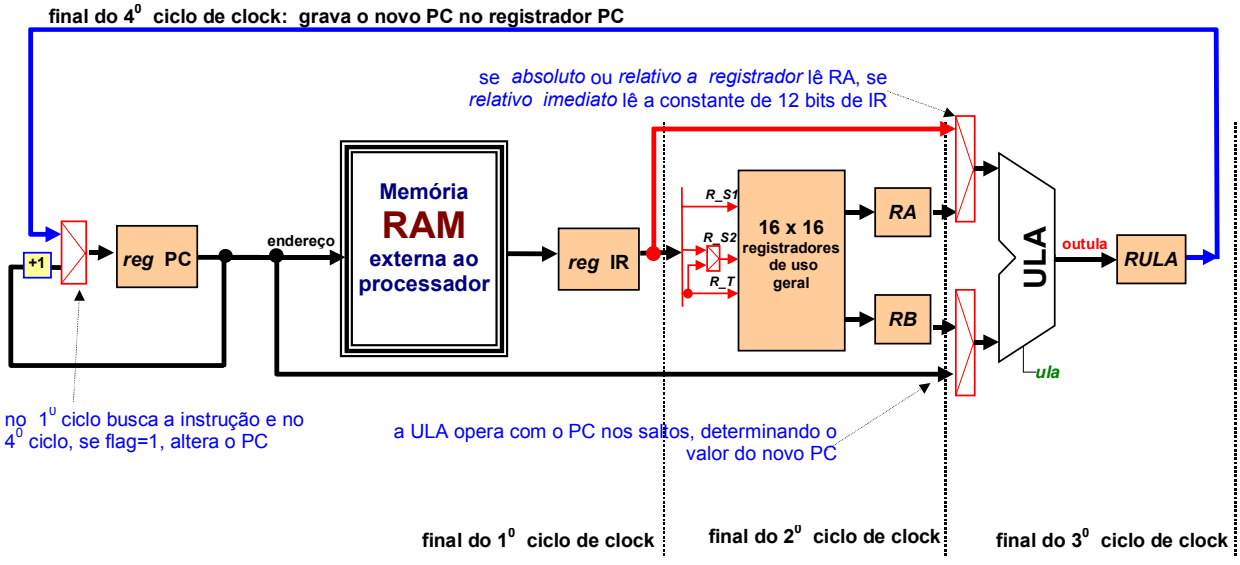
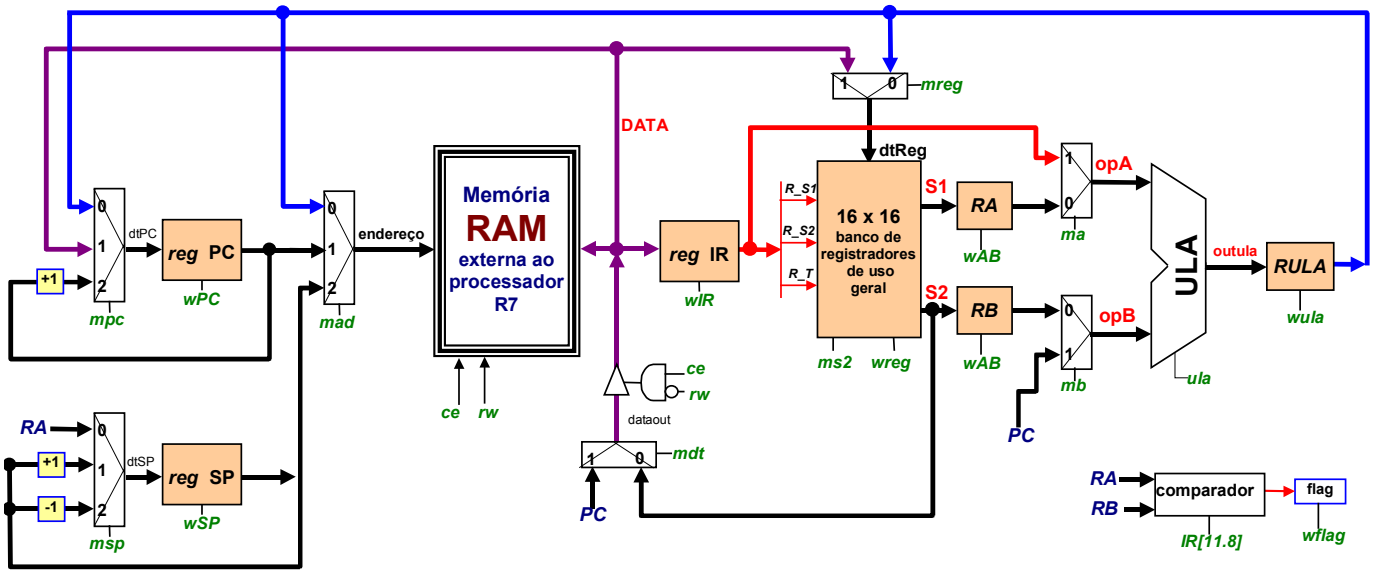


Figura 6 - Fluxo de execução de saltos condicionais.

6 ORGANIZAÇÃO DO BLOCO DE DADOS

Unindo as diversas figuras anteriores, obtemos o diagrama da Figura 7. 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 do flag de estado.



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 7 - Bloco de dados completo (mais memória externa).

A Figura 8 ilustra a organização do banco de registradores, sob forma de um diagrama de blocos.

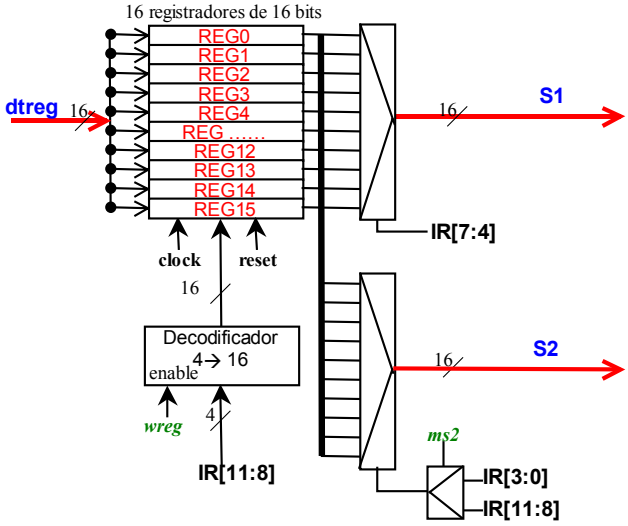
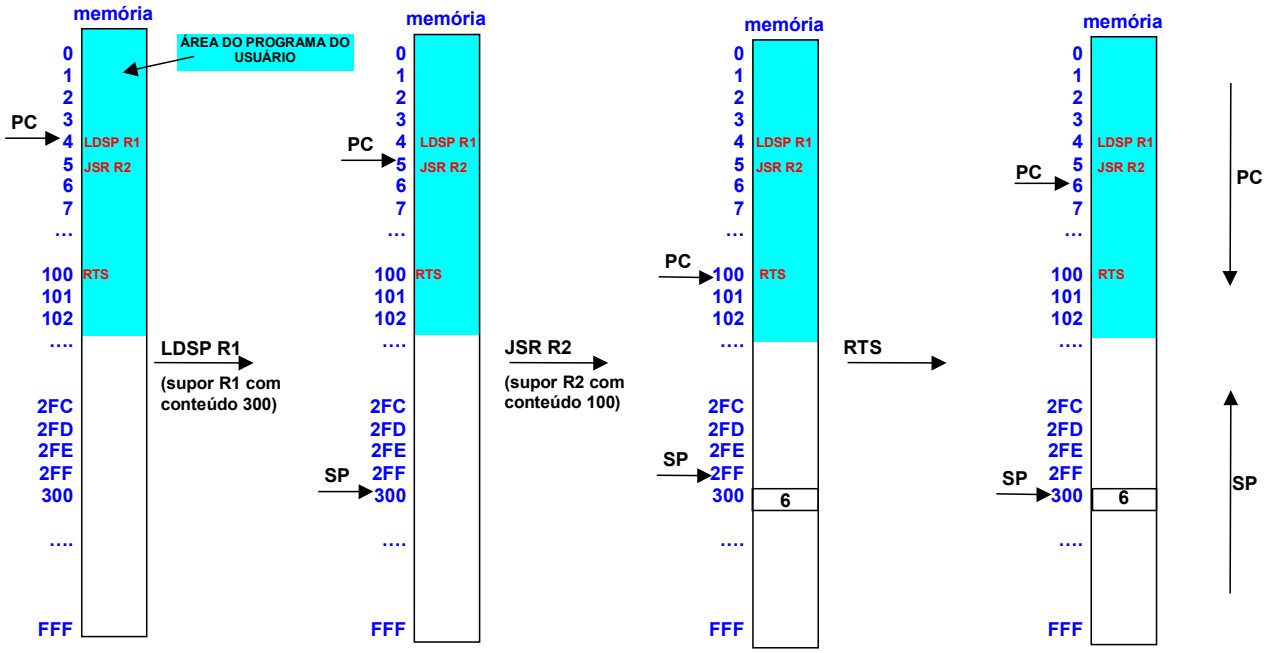


Figura 8 - Diagrama em blocos do banco de registradores de uso geral.

As operações envolvidas com o tratamento de subrotinas são: inicialização do registrador SP (*LDSP*), salto condicional à subrotina (*JSR*, *JSRR*, *JSRD*), e retorno de subrotina (*RTS*).

- A inicialização do SP (LDSP) é feita em três ciclos: *busca, registrador* e armazenamento do conteúdo do registrador RA no registrador SP (observar o multiplexador na entrada do registrador SP).
- A execução de salto para subrotina (se flag=1) é idêntica à execução de salto nos 3 primeiros ciclos. No quarto ciclo várias ações são executadas:
  - atualiza-se o registrador PC com o conteúdo do registrador RULA;
  - grava-se o conteúdo do PC atual na memória (multiplexador com controle *mdt*), endereçando a memória pelo registrador SP (multiplexador com controle *mad*), o que significa gravar no topo da pilha.
  - decrementa-se o conteúdo do registrado SP (multiplexador com controle *msp*), apontando para a nova posição de topo da pilha.
- O retorno de subrotina é executado em 4 ciclos:
  - *busca;*
  - leitura de registradores, não realiza nenhuma operação;
  - incremento do registrador SP (pré-incremento).
  - leitura da memória, gravando o conteúdo endereçada por SP no registrador PC.

A Figura 9 ilustra o funcionamento 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.

Figura 9 - Operação da pilha

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

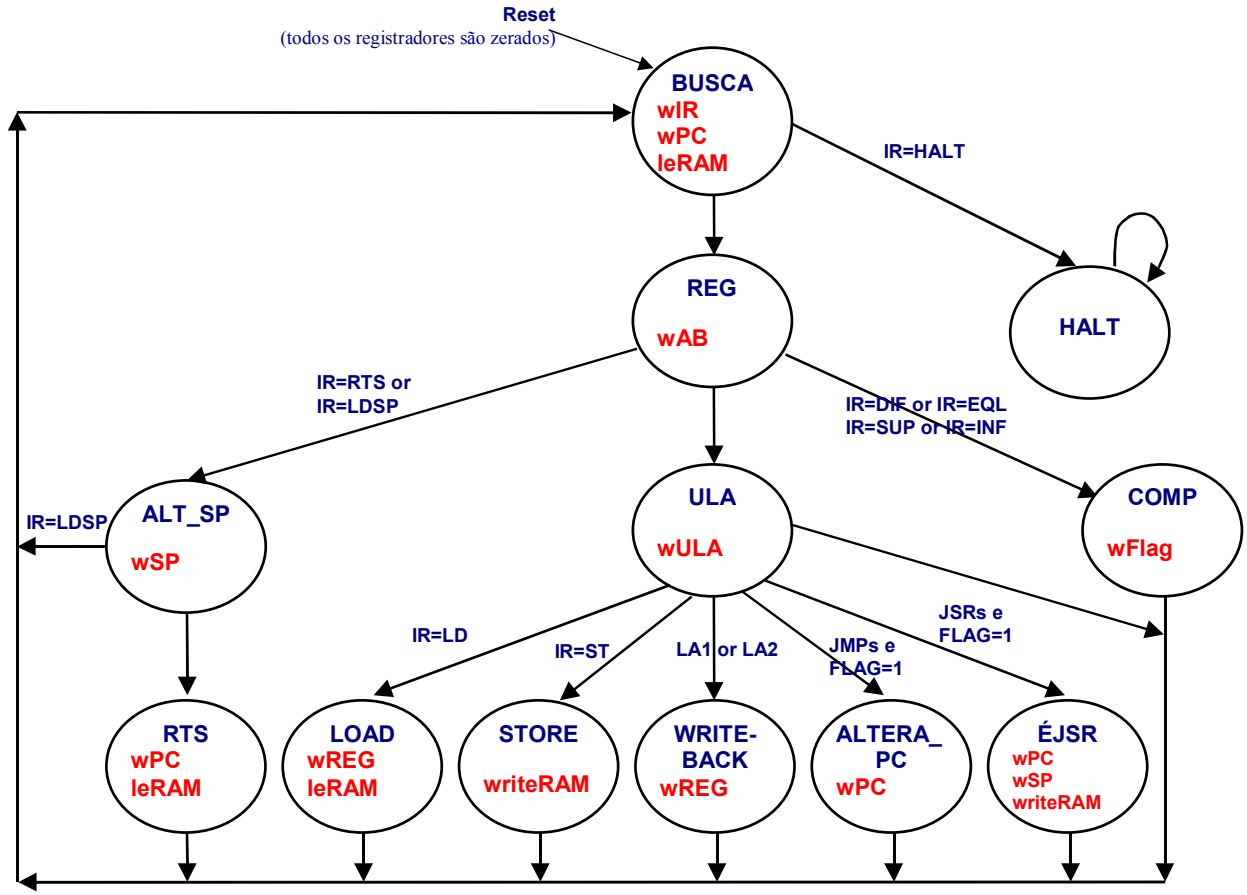
- habilitação de escrita em registradores (7): ***wPC***, ***wSP***, ***wIR***, ***wAB***, ***wULA***, ***wFlag***, ***wReg***.
- controle de leitura/escrita na memória externa (2): ***CE*** e ***RW***.

- controle de multiplexadores (8): ***mpc*** (origem dos dados para o PC), ***msp*** (origem dos dados para o SP), ***mdt*** (qual dado é escrito na memória), ***mad*** (qual registrador endereça a memória), ***mreg*** (origem dos dados para o banco de registradores), ***ms2*** (qual porção 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): ***ula***.

7 BLOCO DE CONTROLE

Observando a execução dos diversos tipos de instrução podemos encontrar 8 estados distintos: *busca*, *registrador*, *ula*, *comp, write-back, load, store* e *alteraPC*. A estes 8 estados, adicionamos 4 outros: *halt* (término da execução), *alt_sp* (que modifica o SP), *rts* (executa as ações finais de retorno de subrotina) e *éjsr* (que executa as ações finais de salto para subrotina).

Desta forma, para executar qualquer instrução neste processador, é necessário definir uma máquina de estados com 12 estados. A Figura 10 ilustra esta máquina de estados, onde o próximo estado é função do estado atual e da instrução armazenada no registrador IR.



LA1 - instrução lógica-aritmética do tipo 1 - operação unária/binária

LA2 - instrução lógica-aritmética do tipo 2 - operação com apenas um registrador fonte

JMPs e JSRs - operações de salto e chamada de subrotina independentes do modo de endereçamento

Figura 10 - Máquina de estados de controle.

Esta máquina de estados controla os 7 sinais de escrita nos registradores e os 2 sinais de acesso à memória (CE e RW).

7.1 Decodificação das instruções

```
i <=  add   when ir(15 downto 12)=0 else
      sub   when ir(15 downto 12)=1 else
      ...
      jmpd  when ir(15 downto 12)=14 else
      jsrd  when ir(15 downto 12)=15;

uins.ula <= i;     -- ** A INSTRUÇÃO DA ULA DEPENDA DA INST. CORRENTE **
```

- Decodificação das instruções lógico-aritméticas do tipo 1:

```
inst_la1 <= '1' when i=add or i=sub or i=e or i=ou or i=oux or i=sl or i=sr or i=notB else
            '0';
```

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

```
inst_la2 <= '1' when i=addi or i=subi or i=ldl or i=ldh else
            '0';
```

- Decodificação das instruções de salto com teste de flag:

```
inst_jmp <= '1' when (i=jmpr or i=jmp or i=jmpd) and flag='1' else
            '0';
```

- Decodificação das instruções de salto à subrotina com teste de flag:

```
inst_jsr <= '1' when (i=jsrr or i=jsr or i=jsrd) and flag='1' else
            '0';
```

DICA - utilizar enumeração (no *package*) para indicar o tipo de instrução a ser decodificada:

```
type instrucao is
    (add, sub, e,   ou,  oux,  addi, subi, ldl,  ldh,  pop, push, ld,   st, sl, sr, notB,
     dif, eql, sup, inf, ldsp, rts,  halt, jmpr, jsrr, jmp, jsr,  jmpd, jsrd);
```

7.2 Controle dos multiplexadores

Os sinais de controle dos multiplexadores (8) dependem do estado da máquina de controle (EA) e/ou da instrução corrente. Recomenda-se localizar os controles dos multiplexadores na Figura 7. Os sinais de controle são precedidos do sufixo "*uins.*", que designa a microinstrução.

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

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

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

```
uins.msp <= "10" when inst_jsr='1' else    -- pos-decremento em chamada à subrotina
            "01" when i=rts        else    -- pre-incremento em retorno de subrotina
            "00";
```

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

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

4. Controle da origem dos dados para escrita na memória (depende apenas da instrução):

```
uins.mdt <= inst_jsr;          -- escreve-se na memória o PC quando houver chamada à subrotina
```

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

```
uins.mreg <= '1' when i=ld else '0';      -- escreve nos registradores o conteúdo vindo da
                                          -- memória na instrução load
```

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

```
uins.ms2 <= '1' when inst_la2='1' or EA=estore else '0';
```

7. Controle da origem dos dados para a ula (depende apenas da instrução):

```
-- primeiro operando da ULA é o IR quando for operação log_arimetica do tipo 2 ou jump/jsr com desloc. Curto
uins.ma <= '1' when inst_la2='1' or i=jmpd or i=jsrd else '0';
```

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

```
uins.mb <= inst_jmp  or inst_jsr;          -- em jumps/jsr o PC será o segundo operando da ULA
```

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

1) Decodificação da instrução: um conjunto de *n* linhas, onde *n* designa o número de instruções do processador, e cada linha é responsável por decodificar uma determinada 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 11 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    BD01      ; grava o conteúdo do registrador D no endereço contido no registrador 1 (190H ou 400)
012D    AF10      ; lê o conteúdo do endereço contido no registrador 1, gravando no registrador 15
```

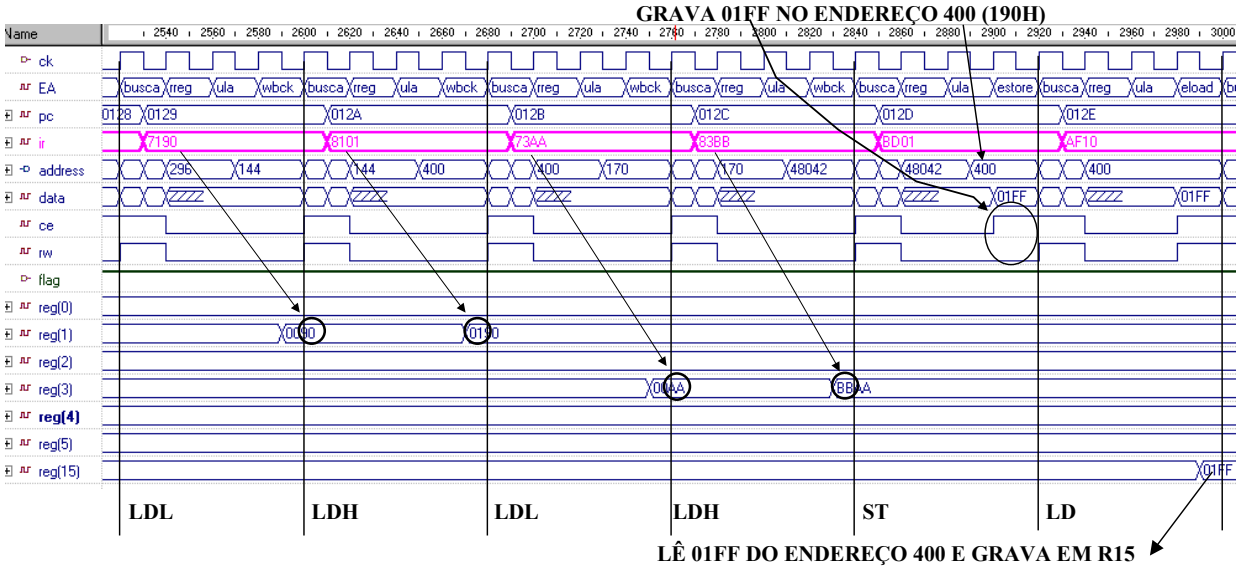


Figura 11 - Simulação de 6 instruções.

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 R7: 023C

000B 5101      ; soma imediato 01 no R1  - 1009
000C 510F      ; soma imediato 0F no R1  - 1018
000D 51FF      ; soma imediato FF no R1  - 1117  (1110 ns)
000E 6201      ; sub. imediato 01 de R2  - 1233
000F 6204      ; sub. imediato 04 de R2  - 122F
0010 62FF      ; sub. imediato FF de R2  - 1130

0011 7DFF      ;
0012 8D01      ;  LOAD RD, #01FF
0013 D4D0      ;  carrega o topo da pilha com o conteúdo do registrador RD(511 em decimal)(1590 ns)
0014 D111      ;  SETA O FLAG comparando R1=R1 ?
0015 7100
0016 8101      ;  R1 <- 0100
0017 DB10      ; ********* salta para subrotina apontada pelo R1 (endereco 0100)  *********

0018 D000      ; RESSETA O FLAG
0019 7730
001A 8700
001B D111      ; SETA O FLAG
001C DA70      ; salta para o endereço contido no registrador 7 (30H)

0030 7710
0031 8700
0032 D870      ; salta relativo para o endereco 33H+10H=43H

0043 E050      ; salto para o endereco 44H+50H=94H

0094 D700      ; *********** HALT HALT *******   (4170 NS DE SIMULACAO)

0100 710D      ; (1890 NS DE SIMULACAO)
0101 8100      ; CARREGA 000D NO REGISTRADOR R1
0102 D910      ; CHAMA OUTRA SUBROTINA - MODO RELATIVO A PC (110H-103H=0DH)
0103 D600      ;  ****** rts **************

0110 4111      ; SUBROTINA QUE zera com xor os registradores 1 a 4 utilizando XOR
0111 4222
0112 4333
0113 4444
0114 F013      ;  subrotina relativo ao PC, sala 13 palavras indo para o endereço 0128
0115 D600      ;  ****** 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 BD01      ; grava o conteúdo do registrador D no endereço contido no registrado 1 (190H ou 400)
012D AF10      ; lê o conteúdo do endereço contido no reg 1, gravando no reg 15    (3250 ns)
012E C101
012F C212
0130 C323      ; testa sl, sr e not
0131 D600      ;  ****** rts ***************** (3250 NS DE SIMULACAO)
```

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 (*geral*, que corresponde ao *flag*) de estado encontram-se na parte inferior à direita.

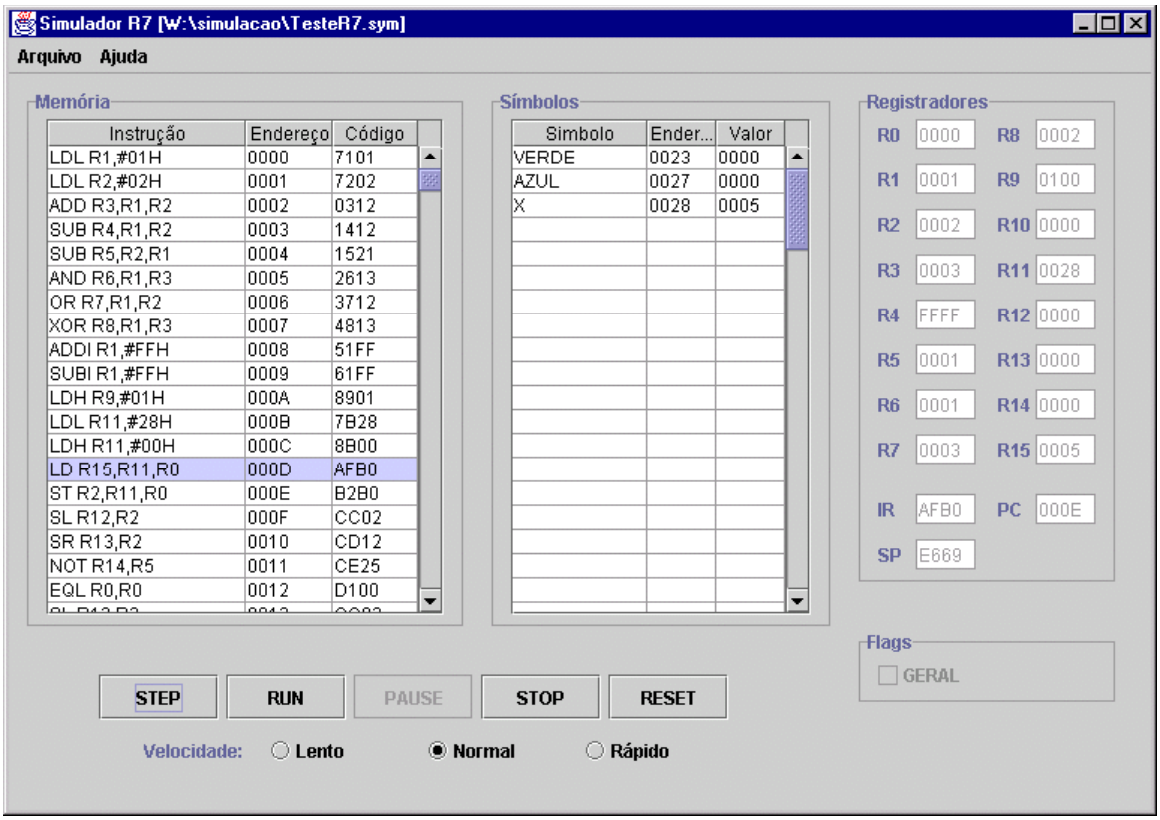


Figura 12 - Simulador R7.

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

10 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 descrito na Seção 7, através de uma máquina de estados. **A nota dará ênfase na execução correta da simulação.**

As regras do trabalho 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.
- 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 assembly).
- 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.
- 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.
- As apresentações ocorrerão na semana ***27-29/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. 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.

- Composição da nota:

BL. DE DADOS	BL. CONTROLE	Estrutura Geral e test_bench	Simulação das instruções básicas	Execução correta de subrotinas
25 %	25 %	5 %	25 %	20 %

Observar que o peso da simulação é de 45%. **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.**

O código assembly da(s) aplicação(ões) desenvolvido deve ser comentado. Recomenda-se entregar um relatório detalhando a implementação e a aplicação.

Importante: o código VHDL deve ser comentado, a fim de facilitar a correção por parte do professor.