Introdução aos processadores programáveis

Henrique Bernardes

26 de outubro de 2025

1 Início - Arquitetura básica de um processador programável

Começamos aqui esclarecendo as diferenças entre hardware dedicado e um processador de propósito geral, tendo como objetivo diferenciar o que estávamos trabalhando do que vamos passar a trabalhar.

Um hardware dedicado implementa circuitos digitais para realizar uma determinada tarefa, podendo assim ser costumizado. Além disso, principalmente pela capacidade de paralelização, hardware dedicado tem um alto desempenho. Contudo, apresenta alto custo e pouca flexibilidade, uma vez que qualquer alteração reflete uma modificação no circuito.

Já um processador de propósito geral tem seu "programa"armazenado em memória, podendo ser modificado, uma vez que se baseia em software, acarretando, também, em um menor tempo de desenvolvimento. Contudo, pelo fato de aqui termos uma sequência de instruções ao invés de um fluxo de dados, é difícil paralelizar o algoritmo, apresentando um baixo desempenho. Dentre os diversos processadores de propósito geral podem ser listados os Intel, AMD, ARM, MIPS e ATMega.

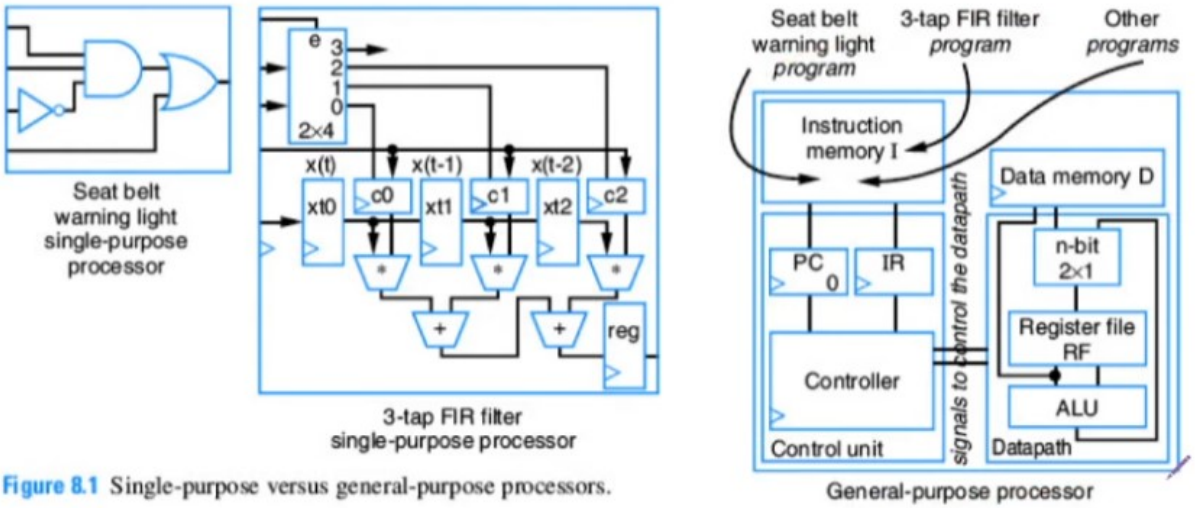


Figure 8.1 Single-purpose versus general-purpose processors.

Figura 1: Hardware dedicado versus processador de propósito geral

2 Arquitetura load-store

2.0.1 Caminho de dados

A memória de dados armazena todos os dados que o processador pode acessar. Esses dados devem passar pelo banco de registradores antes de ser processados pela Unidade Lógica Aritmética(ULA), armazenando os dados a serem operados pela ULA. Ao longo do caminho, o multiplexaddor 2x1 escolhe entre os dados da memória ou os resultados intermediários da ULA.

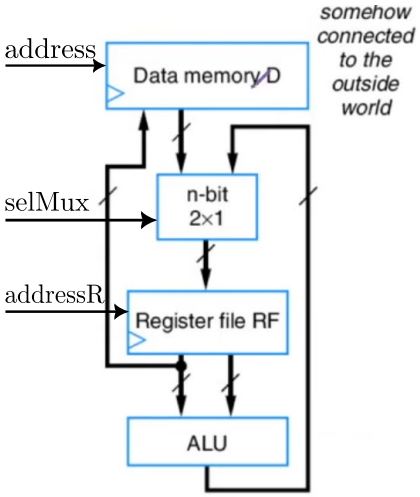


Figura 2: Caminho de dados de uma arquitetura load-store.

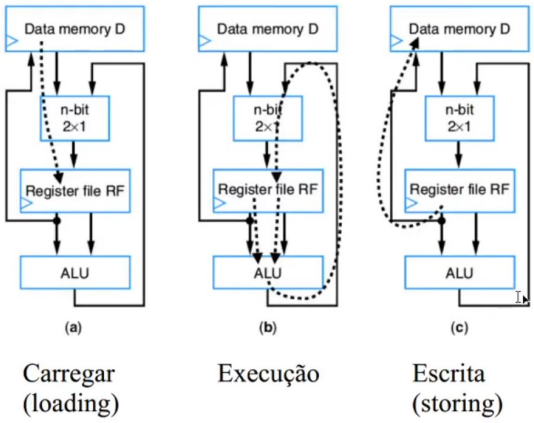


Figura 3: Casos de carregação, execução e escrita de dados em uma arquitetura load-store

Ou seja, o processamento ocorre por meio destas três instruções (figura 3):

1. Carregar dados - leitura de um determinado *address* e carregamento no banco de registradores;
2. Transformar dados - executar operações;
3. Armazenar dados - escrita - um resultado intermediário vindo da ULA pode ser enviada de volta para a memória de dados (também, para o endereço *address*)

2.0.2 Unidade de Controle

A unidade de controle está para o caminho de dados como uma FSM, de fato, como é de se esperar, ela controla o caminho de dados; ou seja, nesta unidade são configurados os sinais de *clear*, *load*, *addresses* da memória de dados e do banco de registradores e seleção do multiplexador.

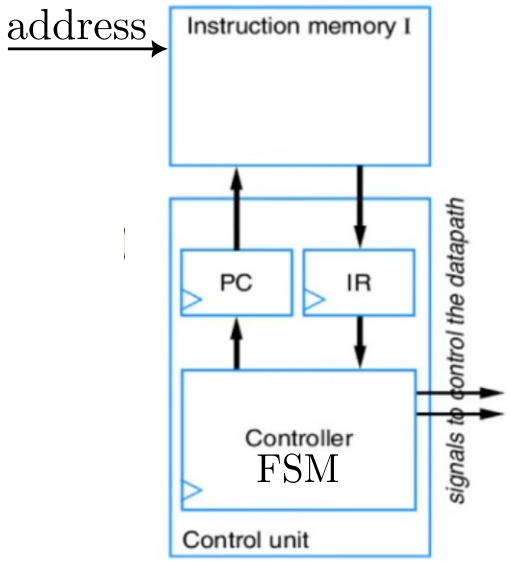


Figura 4: Unidade de controle básica.

Inicialmente, a unidade lê as instruções da memória de instruções I.

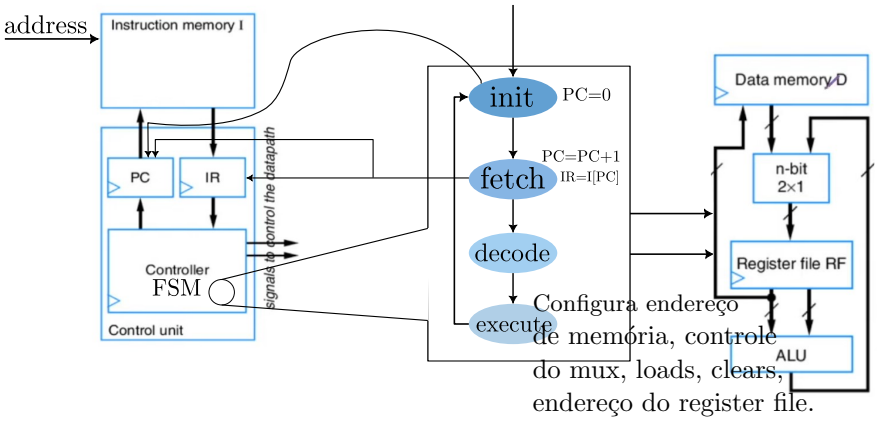
Em uma arquitetura FDx de três estágios, as intruções são realizadas em uma média total de três ciclos de clock, baseadas em três componentes:

- PC (*program counter*): Contador que indica a próxima instrução a ser lida;
- IR (*instruction register*): Armazena a instrução;
- Controlador(*Controller*): FSM

As operações são descritas abaixo:

1. Busca (***F**etch*): Lê I[address] e armazena no registrador local IR(*instruction register*);
2. Decodificação (***D**ecode*): Determina a operação a ser executada;
3. Execução (***Ex**ecute*): Gera os sinais de controle para o caminho de dados de modo que a instrução seja executada.

A figura 5 demonstra a interface entre o *controller*(FSM) e o resto da unidade:



Configura endereço de memória, controle do mux, loads, clears, endereço do register file.

Figura 5: Interface entre os elementos na unidade de controle e a FSM em si.

Temos como exemplo a seguinte operação a ser realizada: $D[9] = D[0] + D[1]$

```
RF[0]=D[0]          – load
RF[1]=D[1]          – load
RF[2]=RF[0]+RF[1]   – (ULA)
D[9]=RF[2]          – write
```

→ Aqui, o valor na posição 2 do register file é escrito na posição 9 do data memory.

Na figura 7 observa-se a primeira linha se propagrando ao longo da unidade de controle e caminho de dados

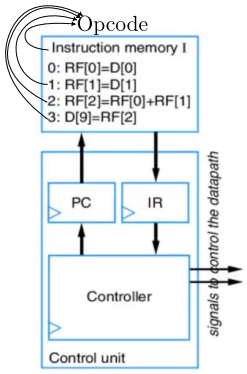


Figura 6: Instruções e armazenamento no *instruction memory I.*

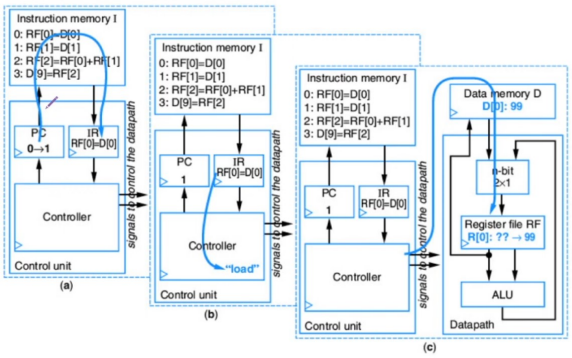


Figura 7: Primeira linha de código se propagando pela unidade de controle e caminho de dados.

3 Processador programável de três instruções

Inicialmente, definimos o que é conhecido como **conjunto de instruções** do processador programável: é uma lista das instruções possíveis e a meneira de representar essas instruções na memória. Vamos definir, também, que um processador usa instruções de 16 bits e que a memória de instruções I tem 16 bits de largura. Normalmente, um determinado número de bits é reservado pelo conjunto de instruções para indicar a instrução em si a ser realizada sendo que os restantes especificam informações adicionais como registradores de origem, destino, etc.

Definiremos um conjunto de três operações símples, onde os 4 bits mais significativos irão representar a operação e os 12 restantes indicam os endereços no banco de registradores(*register file*) e na memória de dados.

Teremos, então, as seguintes instruções:

```
(0) Carregar - load - 0000(r3)(r2)(r1)(r0)(d7)(d6)(d5)(d4)(d3)(d2)(d1)(d0)
(1) 0000 0000 00000000
(2) 0000 0001 00101010
```

Esta instrução especifica uma movimentação de dados que vai de uma posição da memória de dados(especificado pelos d_i bits) até um registrador no banco de registradores(especificado pelos r_i bits), ou seja, funciona como uma instrução de *fetch*.
Como exemplo, primeiro, a linha 1 indica uma operação de movimentação de dados do da posição 0 da memória de dados($D[0]$) até a posição 0 do banco de registradores($RF[0]$), ou seja, esta instrução realiza a operação $RF[0] = D[0]$.
A linha 2 indica uma operação de movimentação de dados da posição 42 da memória de dados ($D[42]$) até a posição 1 do banco de registradores ($RF[1]$). Ou seja, especifica a operação $RF[1] = D[42]$

```
(0) Armazenar - store -0001(r3)(r2)(r1)(r0)(d7)(d6)(d5)(d4)(d3)(d2)(d1)(d0)
(1) 0001 0000 00001001
```

Essa instrução especifica uma movimentação de dados no sentido oposto da instrução anterior, indo do banco de registradores até a memória de dados, ou seja, funciona como uma instrução de *store*.
Assim sendo, a instrução da linha 1 armazena no endereço 9 da memória de dados ($D[9]$) o dado que está localizado no endereço 0 do banco de registradores($RF[0]$).

```
(0) Somar-add-0010(ra3)(ra2)(ra1)(ra0)(rb3)(rb2)(rb1)(rb0)(rc3)(rc2)(rc1)(rc0)
(1) 0010 0010 0000 0001
```

Essa instrução especifica somar os conteúdos de dois registradores do banco de registradores, especificados por $rb_3rb_2rb_1rb_0$ e $rc_3rc_2rc_1rc_0$ e então armazenar no registrador do banco de registradores especificado por $ra_3ra_2ra_1ra_0$.
Assim sendo, a instrução da linha 1 especifica somar o conteúdo do registrador $RF[0]$ ao conteúdo do registrador $RF[1]$ e armazenar no registrador $RF[2]$, ou seja, $RF[2] = RF[0] + RF[1]$.

Nenhuma dessas operações modifica os conteúdos dos operandos de origem, ou seja:

- A instrução **carregar** faz com que os conteúdos de um endereço da memória de dados $D[i]$ sejam carregados em um registrador do banco de registrador $RF[i]$, sem alterar o conteúdo naquele endereço da memória de dados $D[i]$.

- A instrução **armazenar** faz com que os conteúdos de um endereço do banco de registrador $RF[i]$ sejam armazenados em um endereço da memória de dados $D[i]$ sem alterar o conteúdo daquele registrador $RF[i]$.
- A instrução **somar** faz com que os dados dos registradores $Rb[i]$ e $Rc[i]$ sejam somados e armazenados em $Ra[i]$ sem alterar os conteúdos dos registradores $Rb[i]$ e $Rc[i]$.

Assim sendo, é possível entender o processo para realizar a operação $D[9] = D[0] + D[1]$ (figura 8):

OPERAÇÃO D[9]=D[0]+D[1]

```
(0) load - RF[0] = D[0] - 0000 0000 00000000
(1) load - RF[1] = D[1] - 0000 0001 00000001
(2) add - RF[2]=RF[0]+RF[1] - 0010 0010 0000 0001
(3) store - D[9] = RF[2] - 0001 0010 00001001
```

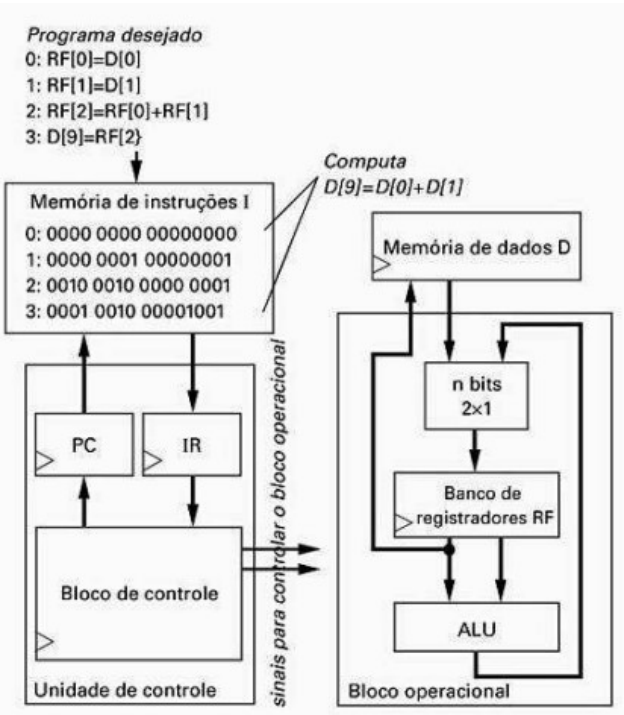


Figura 8: $D[9] = D[0] + D[1]$

Realizando a operação $D[5] = D[5] + D[6] + D[7]$:

```
OPERAÇÃO D[5] = D[5] + D[6] + D[7]:

(0) 0000 0000 00000101   // load  -  RF[0] = D[5]
(1) 0000 0001 00000110   // load  -  RF[1] = D[6]
(2) 0000 0010 00000111   // load  -  RF[2] = D[7]
(3) 0010 0011 0000 0001  // sum   -  RF[3] = RF[0]+RF[1] ou D[5]+D[6]
(4) 0010 0011 0011 0010  // sum   -  RF[3] = RF[3]+RF[2] ou D[5]+D[6]+D[7]
(5) 0001 0011 00000101   // store -  D[5]  = RF[3]

otimizando para utilizar apenas 3 registradores do banco de registradores

(0) 0000 0000 00000101   // load  -  RF[0] = D[5]
(1) 0000 0001 00000110   // load  -  RF[1] = D[6]
(2) 0000 0010 00000111   // load  -  RF[2] = D[7]
(3) 0010 0011 0000 0001  // sum   -  RF[0] = RF[0]+RF[1] ou D[5]+D[6]
(4) 0010 0011 0011 0010  // sum   -  RF[0] = RF[0]+RF[2] ou D[5]+D[6]+D[7]
(5) 0001 0011 00000101   // store -  D[5]  = RF[0]
```

Finalmente, como foi visto, na memórias de instruções, as instruções existem como sequências de 0s e 1s, representação denominada como **código de máquina**. Como é de se esperar, escrever um programa inteiro dessa forma acarretaria em muitos erros. Por esse motivo, utiliza-se uma ferramenta chamada *assembler*, que permite escrever instruções usando mnemônicos, ou símbolos, que o *assembler* traduz para código máquina. Um código escrito em mnemônicos é chamado de **código assembly**. Dessa forma, no contexto do conjunto de três instruções, um *assembler* nos permite escrever as instruções usando os seguintes mnemônicos:

1. Carregar - MOV Ra, d - especifica a operação $RF[a] = D[d]$. O valor de a pode assumir quaisquer valor de 0 a $2^4 - 1$, ou seja, 0 a 15, assim, R0 significa $RF[0]$, R1 significa $RF[1]$, etc. O valor de d pode assumir quaisquer valor de 0 a $2^8 - 1$, ou seja, 0 à 255.
2. Armazenar - MOV d, Ra - especifica a operação $D[d] = RF[a]$;
3. Somar - ADD Ra, Rb, Rc - especifica a operação $RF[a] = RF[b] + RF[c]$

Com o uso desses mnemônicos o programa $D[9] = D[0] + D[1]$ pode ser rescrito como:

```
OPERAÇÃO D[9]=D[0]+D[1]:

(0) MOV R0, 0
(1) MOV R1, 1
(2) ADD R0, R0, R1
(3) MOV 9, R0
```

Tendo estabelecido a base dos processadores de três instruções, como suas instruções e a arquitetura básica, podemos expandir o bloco fsm da figura 5, principalmente, o estado *execute*.

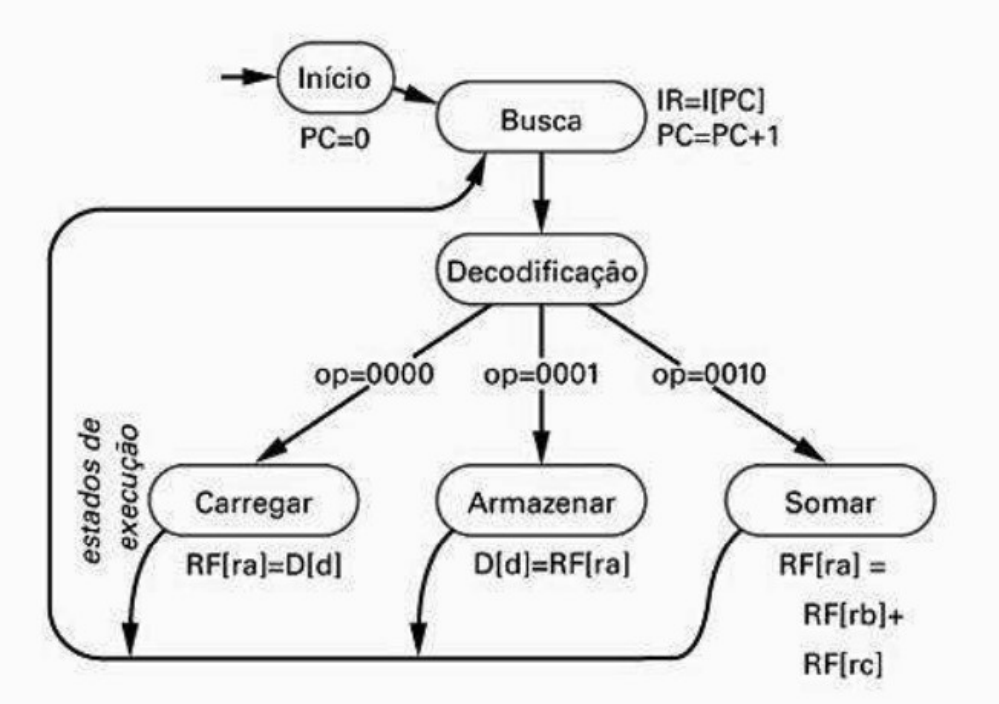


Figura 9: FSM com estado execute expandido.

Primeiramente, assumimos que op é uma abreviação de $IR[15...12]$, ou seja, os bits mais significativos do *instruction register*. O mesmo pode ser dito sobre ra($IR[11...8]$), rb($IR[7...4]$), rc($IR[3...0]$) e d($IR[7...0]$), ou seja:

- op $\rightarrow IR[15...12]$
- ra $\rightarrow IR[11...8]$
- rb $\rightarrow IR[7...4]$
- rc $\rightarrow IR[3...0]$
- d $\rightarrow IR[7...0]$

Feito isso, passamos para o bloco de operacional, agora, mostrando com mais detalhes a integração entre ele e a unidade de controle:

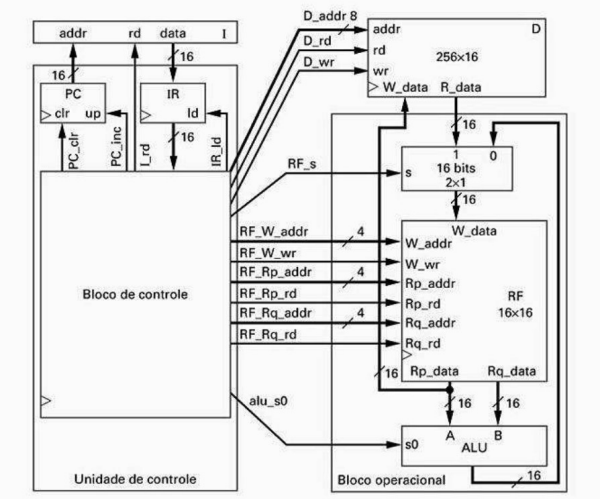


Figura 10: Unidade de controle e bloco operacional e interfaces entra ambos para um processador de três instruções.

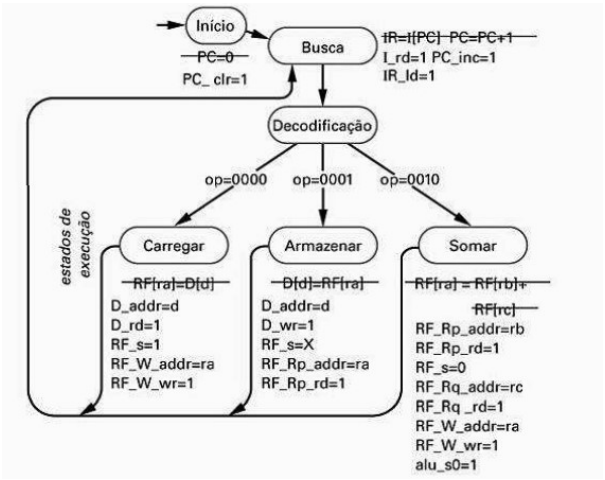


Figura 11: FSM de baixo nível do processador de três instruções.

Por cada instrução possuir 4 bits para endereçar o banco de registradores, ele dispõe de 16 registradores. Também, o bloco operacional tem um sinal de controle para a ALU(ULA) chamado alu_s0. Quando alu_s0=1, a ULA soma as duas entradas; quando alu_s0=0, ela simplesmente deixa passar a entrada. A memória de dados tem 256 palavras, uma vez que cada instrução possui 8 bits para endereçar a memória de dados.

Finalmente, podemos obter a FSM de baixo nível, substituindo as ações de alto nível pelas operações booleanas de saída, vista na figura 11.

Analisando o comportamento da FSM podemos ver como um programa seria executado nesta arquitetura de processador de três instruções:

1. A FSM naturalmente começa no estado Início, com $PC\_clr = 1$, limpando o *program counter* deixando-o em 0;

2. No próximo ciclo de clock, a FSM entra no estado de busca, no qual lê a memória de instruções no endereço 0(porque $PC = 0$) e carrega o valor lido em IR. Ao fim, incrementa o valor de PC. *Algo como* $IR[PC] = IM[PC]$;

3. Em seguida, a FSM entra no estado de decodificação, sem ações realizadas, estando aqui para permitir que os valores dos códigos de operação se atualizem(as atualizações de registrador atribuídas a um estado só ocorrem de fato na próxima borda de clock). A depender dos 4 bits mais significativos do *instruction register*, o próximo estado será carregar, armazenar ou somar;

4. No estado de carregar, a FSM prepara as linhas de endereço de memória de dados utilizando os 8 bits menos significativos do *instruction register*)($D_addr = d$) e habilitando a leitura da memória de dados($D\_rd = 1$). Aqui, a seletora do multiplexador é ativada($RF_s = 1$) para que o valor de saída da memória de dados vá para o banco de registradores. Finalmente, prepara o endereço de escrita do banco de registradores ($RF\_W\_addr = ra$), usando $IR[11...8]$ e habilitando a escrita ($RF\_W\_wr = 1$). Tudo isso faz com que o conteúdo que esteja na memória de dados seja carregado no registrador apropriado do banco de registradores.

5. De forma semelhante os estados Armazenar e Somar preparam as linhas de controle conforme a necessidade das operações de armazenamento e soma.

6. Por fim, a FSM retorna ao estado de Busca, buscando pela próxima instrução.

4 Processador programável de seis instruções

Tendo em vista as limitações de um processador de três instruções, estendemos o conjunto de instruções do processador acrescentando mais três instruções. Primeiramente, além das instruções vindas do processador de três instruções, *load*, *store* e *add*, uma nova instrução permitirá carregar uma constante para um dos registradores do banco de registradores, permitindo novas operações como $RF[0] = RF[1] + 5$, onde 5 é uma constante, ou seja,não se encontra na memória de dados.

Outra operação nova será a de subtração: similar à operação de soma já existente.

Finalmente, a sexta e última operação será a de "Saltar se zero". Esta instrução permitirá saltar para alguma parte do programa caso o conteúdo de algum registrador especificado for 0.

Expandindo nestas operações:

Carregar constante – 0011(r3)(r2)(r1)(r0)(c7)(c6)(c5)(c4)(c3)(c2)(c1)(c0)

Especifica que o número binário representado pelos bits c_i(conhecido como uma constante) deve ser deve ser carregado no registrador especificado pelos bits r_i. Aqui, r_i corresponde à IR[11...8] e c_i corresponde à IR[7...0]. O mnemônico dessa instrução é:

```
MOV Ra, #c // RF[a]=c
```

onde a pode assumir qualquer valor de 0 até $2^4 - 1$ em complemento de 2, ou seja, de 0 até 15, e c pode assumir qualquer valor entre $\frac{-2^8}{2}$ e $\frac{2^8-1}{2}$, ou seja, entre -128 e $+127$.

Subtrair – 0100(ra3)(ra2)(ra1)(ra0)(rb3)(rb2)(rb1)(rb0)(rc3)(rc2)(rc1)(rc0)

Especifica a subtração dos conteúdos de dois registradores do banco de registradores especificados por rb_i e rc_i. O resultando é então armazenado no registrador do banco de registradores especificado por ra_i(Lembrando que ra corresponde IR[11...8], rb corresponde IR[7...4] e rc corresponde IR[3...0]). O mnemônico dessa instrução é

```
Sub Ra, Rb, Rc // RF[a]=RF[b]+RF[c]
```

Os termos a, b e c podem assumir qualquer valor de 0 até $2^4 - 1$ em complemento de 2, ou seja, de 0 até 15.

Saltar se zero – 0101(ra3)(ra2)(ra1)(ra0)(o7)(o6)(o5)(o4)(o3)(o2)(o1)(o0)

Especifica que se o conteúdo do registrador especificado por ra_i for 0, então carrega-se o PC com o valor atual de PC somado ao valor em oito bits e complemento de 2 o_i, podendo ser um *offset* positivo ou negativo. Aqui, ra_i corresponde à IR[11...8] e o_i à IR[7...0]. O mnemônico dessa instrução é:

```
JMPZ Ra, offset // PC = PC + offset se RF[a]=0
```

Offset pode tomar qualquer valor entre $\frac{-2^8}{2}$ e $\frac{2^8-1}{2}$. Ou seja. podemos saltar para frente até 127 endereços ou para trás até 128 endereços. Aqui, também, onde a pode assumir qualquer valor de 0 até $2^4 - 1$.

Resumindo, em um processador programável de seis instruções temos as seguintes instruções:

Tabela 1: Conjunto de instruções de seis instruções

Instrução	**Significado**
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b] + RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b] - RF[c]
JMPZ Ra, offset	PC = PC + offset se RF[a] = 0

Devemos também estender a noção da unidade de controle e bloco operacional para o processador de seis instruções)(figura 12). Primeiramente, a instrução de carregar constante necessita que possamos carregar o valor de $IR[7...0]$ no banco de registradores, além dos valores da memória de dados e da saída da ULA. Assim, aumenta-se o multiplexador do banco de registradores de 2x1 para 3x1, adiciona-se também mais um sinal de controle ao multiplexador e cria-se um sinal novo denominado RF_w_data que vai do bloco de controle se ligando à $IR[7...0]$ e entrano no multiplexador.

Segundo, a instrução subtrair requer que a ULA seja capaz de realizar a subtração. Então, adiciona-se um novo sinal de controle à ULA visando diferenciar quando fazer uma subtração e quando fazer uma adição, alu_s1.

Por fim, a instrução saltar se zero requer a capacidade de detectar se um registrador é zero e somar $IR[7...0]$ ao PC. Assim, insere-se um componente no bloco operacional para determinar se a porta de leitura Rp do banco de registradores es´ta com todos os bits em zero. Também, modifica-se PC para que seja possível carregar $PC = PC + IR[7...0] \therefore PC = PC + offset$. Vale mencionar que o somador utilizado já subtrai 1 da soma, compensando o fato que o estado de Busca já incrementa 1 ao PC.

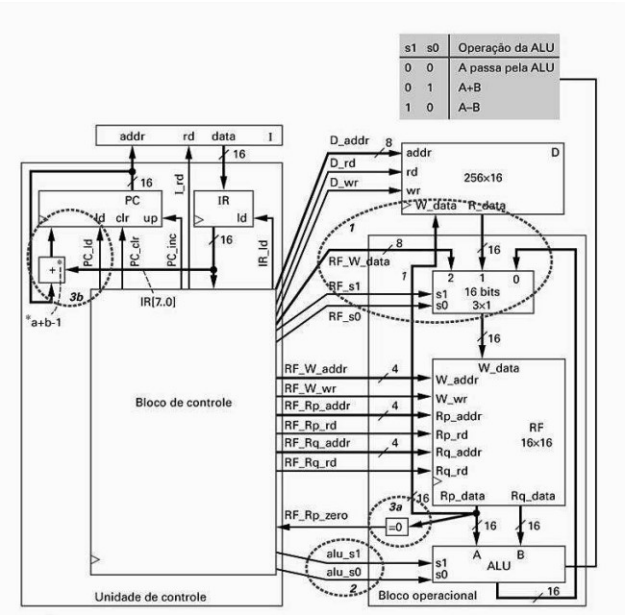


Figura 12: Unidade de controle e bloco operacional e interfaces entra ambos para um processador de seis instruções.

Vamos estender também, naturalmente, a FSM do bloco de controle para lidar com as instruções adicionais. A figura 13 mostra seu diagrama:

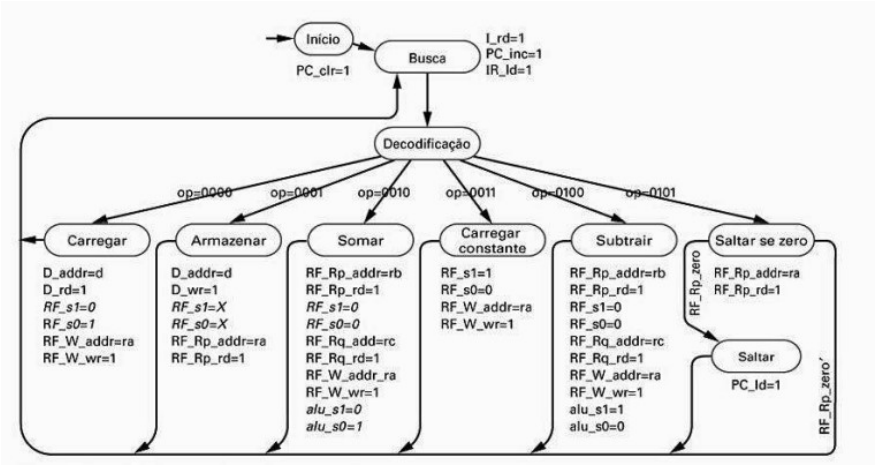


Figura 13: FSM de baixo nível do processador de seis instruções

Os estados início e busca permanecem os mesmos. Em decodificação, adiciona-se três novas transições para as três novas instruções. Também, os estados carregar, armazenar e somar sofreram alterações para refletir a necessidade do multiplexador do banco de registradores possuir duas variáveis de seleção e a ULA ser configurada com dois sinais. Além disso, foram adicionados os três estados extras:

1. No estado Carregar constante, configura-se o multiplexador do banco de registradores para deixar passar o sinal RF_W_data e o banco de registradores para escrever no endereço especificado por ra($IR[11...8]$);

2. No estado Subtrair, executa-se as mesmas ações do estado Somar, exceto que a ULA é configurada para subtração ao invés de adição ($s1 = 1$ e ($s0 = 0$);

3. No estado de Saltar se zero, configura-se o banco de registradores para que o registrador especificado por ra seja lido e seu conteúdo colocado na port ade leitura Rp. Se o valor de Rp for composto apenas de 0s, então a saída RF_Rp_zero irá ser 1(e 0, caso contrário). Dessa forma, deve-se tratar de duas transições partindo do estado Saltar se zero. Se $RF\_Rp\_zero = 0$, ou seja, o registrador lido(presente agora na saída Rp) não contém apenas 0s, a FSM volta para o estado Busca, ou seja, não ocorre salto. Se $RF\_Rp\_zero = 1$, o registrador lido é composto só de zeros, então a FSM vai para o próximo estado, Saltar, na qual realmente ocorro o salto com o sinal de load do PC sendo 1($PC\_ld = 1$).