# Modelação em C++, Síntese e Implementação de Circuitos Digitais com base em FPGA

Valery Sklyarov

***Resumo*** **– Este artigo apresenta uma técnica de desenvolvimento de circuitos digitais que pode ser utilizada por alunos de disciplinas de programação orientada por objectos, computação reconfigurável, sistemas digitais avançados, etc. A técnica é baseada em *hardware templates* (HT) que são circuitos desenvolvidos para um grupo de aplicações semelhantes tais como processadores de uso especial, controladores embutidos, etc. A personalização do HT para uma aplicação particular é conseguida através da especificação da sequência de controlo apropriada. Alterações na sequência de controlo podem ser efectuadas sobre circuitos de controlo que permitem a modificação estática e dinâmica do seu comportamento. O comportamento é especificado com a ajuda de máquinas de estados finitos reprogramáveis (MEFR). Assume-se que, para o grupo de aplicações considerado, o HT já terá sido desenvolvido, modelado em C++ e implementado em FPGA. A sequência de projecto é composta por passos seguintes: 1) especificação e modelação da funcionalidade desejada em C++; 2) transformação de funções em C++ que descrevem várias sequências de operações de controlo, no modelo de software dos circuitos de hardware respectivos; 3) síntese de *bitstreams* para a MEFR (ou para um conjunto de MEFRs interligadas); 4) implementação do circuito com base em FPGA. O artigo apresenta um exemplo que permite realizar computações simples sobre vectores e matrizes booleanas e ternárias, e apresenta em detalhe todos os passos mencionados acima.**

***Abstract*** **– The paper presents a technique for the design of digital circuits that can be employed by students within such disciplines as object-oriented programming, reconfigurable computing, advanced digital systems, etc. The technique is based on the use of a hardware template (HT), which is a circuit that has been designed for a wide group of similar applications, such as special-purpose processors, embedded controllers, etc. Customizing the HT for a particular application is achieved by specification of the proper control sequence. Possible changes in control sequences might be carried out by control circuits that allow static and dynamic modifications to their behavior. Such behavior is provided with the aid of a reprogrammable finite state machine (RFSM). It is assumed that HT for the considered group of applications has already been constructed, modeled in C++, and implemented in FPGA. The considered design flow is composed of the following steps: 1) specification and modeling of the desired functionality in C++; 2) translation of C++ functions that describe various sequences of control operations to software model of the respective hardware circuits; 3) synthesis of bitstreams for RFSM (or possibly for a set of communicating RFSMs); 4) implementation of the circuit on the base of FPGA. The paper shows a design example for simple computations over Boolean and ternary vectors and matrices and demonstrates all the considered above steps.**

## I. INTRODUCÃO

Os sistemas digitais podem ser divididos em duas unidades que são a unidade de controlo (UC) e a unidade de execução (UE). A UC estabelece a sequência de operações que devem ser executadas pela UE (ver fig. 1). Por outras palavras, a UC implementa um algoritmo de controlo que define esta sequência, i.e. a UC gera os sinais de saída (que especificam as operações a executar pela UE) em dependência dos de entrada (que representam alguns estados da UE). Para especificar um algoritmo de controlo pode-se utilizar linguagens tais como "*graph-schemes of algorithms*" (GS) [1] e "*hierarchical* GS" (HGS) [2], com várias extensões apresentadas em [3,4,5]. A fig. 2 mostra um exemplo de especificação dum algoritmo de controlo com base em GS, onde $Y=\{y_1,...,y_N\}$ (N=5) são as saídas (microoperações - MO) da UC e as entradas da UE, e $X=\{x_1,...,x_L\}$ (L=2) (condições lógicas - LC) são as entradas da UC e as saídas da UE (ver fig. 1). De notar que GS e HGS se baseiam no formalismo da máquina de estados finitos (MEF) e cada especificação pode ser convertida no respectivo grafo que descreve o comportamento da MEF [1,2].

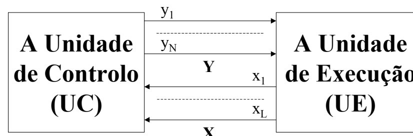


Fig. 1 - Interligação entre UC e UE

Normalmente, a UE pode ser considerada como um conjunto de registos e barramentos que interligam os registos e disponibilizam vários caminhos para transferir dados (*register transfer specification*). Na fig. 3 está representado um exemplo de uma UE composta por

quatro componentes: os contadores (*counter* e *ccv*), um registo (*size*) e um registo de deslocação (*vector*). As entradas **I1** e **I2** permitem inicializar a UE, i.e. gravar no registo *vector* um vector binário e no registo *size* o tamanho do vector. A UE pode ser controlada através das MOs Y={$y_1$,...,$y_5$} cujos sinais correspondentes provêm das saídas da UC (ver fig. 1). Os sinais X={$x_1$,$x_2$} gerados pela UE representam os estados dos componentes indicados acima e permitem efectuar as transições condicionais da UC (ver fig. 1).

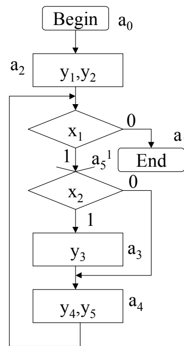


Fig. 2 - Um exemplo de GS

A UE pode ter uma arquitectura universal tal como a usada para um microprocessador geral, ou especial, permitindo resolver problemas específicos, por exemplo na área de computação sobre matrizes binárias e ternárias [6]. De notar que em muitas aplicações cada tarefa requer os algoritmos específicos a serem implementados na UC. Por outras palavras, vários problemas da mesma área requerem algoritmos diferentes que, por outro lado, podem ser executados na mesma UE. Então a decomposição básica (ver fig. 1) pode ser considerada como a interligação entre uma UE orientada para um problema e uma UC com um comportamento reprogramável.

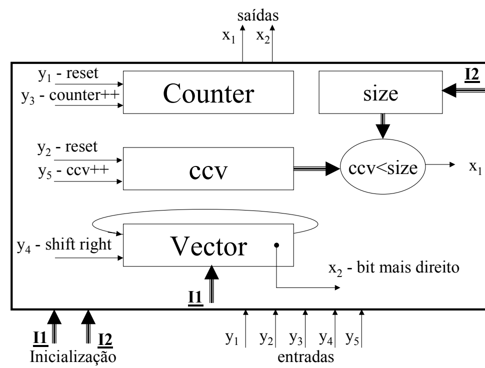


Fig. 3 - Um exemplo de UE

Neste artigo mostra-se como construir circuitos digitais orientados para problemas específicos com base nesta decomposição. O desenvolvimento do respectivo projecto inclui os seguintes passos:

1) a apresentação, com base nas classes da linguagem C++, do modelo geral da arquitectura que pode ser reutilizada para resolver vários problemas de uma área específica. Vamos assumir que este modelo pode ser implementado em hardware (em FPGA em particular);

2) com base na arquitectura considerada acima, a implementação e a verificação de software para resolver os problemas concretos da respectiva área;

3) a conversão das funções de C++ que definem o comportamento, para os GS ou HGS dos algoritmos especificados;

4) a modelação e a verificação dos algoritmos em software com a ajuda das funções da linguagem C++ que especificam directamente as operações de hardware;

5) a síntese de *bitstreams* para a MEFR que permitem implementar os algoritmos de controlo considerados acima;

6) a implementação da MEFR em hardware (em FPGA).

A organização da parte restante deste artigo integra mais 6 secções. Na secção II são discutidas as unidades de controlo reprogramáveis. Na secção III é demonstrado como implementar circuitos de controlo reprogramáveis com base em FPGAs. Na secção IV descreve-se um modelo orientado por objectos da unidade de controlo reprogramável. A secção V introduz o modelo orientado por objectos que permite simular a interligação ente a UE e a UC. A seguir, na secção VI, apresenta-se a simulação em C++ e o desenvolvimento de circuitos digitais com base em hardware templates. Finalmente, as conclusões estão incluídas na secção VII.

## II. UNIDADE DE CONTOLO REPROGRAMÁVEL

A fig. 4 mostra uma arquitectura da unidade de controlo reprogramável (UCR) composta por três blocos: um registo (Rg), um multiplexador reprogamável (PM -

*programmable multiplexer*) e a memória que implementa as transições entre os estados da UCR (STRAM - *state transition RAM*). O modelo da UCR é uma MEFR [7]. O PM (ver fig. 5) permite seleccionar qualquer entrada $x_l \in X = \{x_1,...,x_L\}$ e ligar esta entrada à saída p (por isso $p=x_l$). O índice l pode ser especificado por programação da MRAM (*multiplexer* RAM) que faz parte do PM. Por exemplo, a fig. 5 mostra como programar a MRAM para

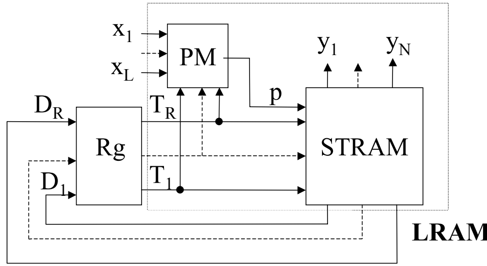


Fig. 4 - Arquitectura inicial da UCR

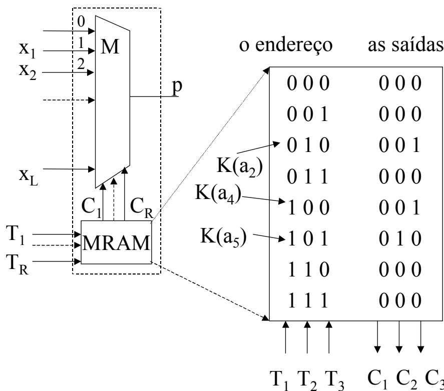


Fig. 5 - A estrutura do PM

Contudo, o circuito na fig. 4 tem um problema. Cada transição de estados não pode ser influenciada por mais que uma condição lógica $x_l \in X$ (i.e. mais que uma entrada). Consideremos um exemplo. É conhecido que um GS pode ser convertido na tabela habitual da MEF (ou no grafo da MEF). Por exemplo, o GS na fig. 2 pode ser marcado com os estados $a_0,a_1,...,a_4$ (o estado $a_0$ foi atribuído ao nó "*Begin*", o estado $a_1$ - ao nó "*End*", os estados $a_2,...,a_4$ - aos nós rectangulares do GS). A tabela 1 mostra todas as transições entre os estados, onde $a_{from}$ é o estado inicial na transição, $K(a_{from})$ é o código do estado inicial, $a_{to}$ é o estado seguinte na transição, $K(a_{to})$ é o código do estado seguinte, $Y(a_{from})$ é o conjunto de saídas

escolher a entrada $x_1$ nos estados $a_2$ e $a_4$ com os códigos $K(a_2)$=010 e $K(a_4)$=100 e a entrada $x_2$ no estado $a_5$ com o código $K(a_5)$=101. Como resultado nos estados $a_2$ e $a_4$ temos $p=x_1$ e no estado $a_5$ - $p=x_2$.

Obviamente pode-se estabelecer qualquer associação entre os estados da MEF $a_0,...,a_{M-1}$ e as entradas $x_1,...,x_L$. A STRAM permite gerar os códigos para as saídas da MEF e para os estados seguintes.

com valor 1 no estado $a_{from}$. Para a simplificação, os códigos dos estados têm o valor binário do índice do estado respectivo. Se algumas transições forem causadas por mais do que uma condição lógica (ver transições dos estados $a_2$ e $a_4$) é impossível utilizar o circuito da fig. 4. Por outro lado, a tabela poder ser convertida noutra tabela (ver tabela 2) que não contém transições causadas por mais que uma entrada. Isto pode ser feito com a ajuda de estados adicionais tais como $a_5^1$ (ver fig. 2). Para este fim podemos utilizar o método considerado em [8]. Um fragmento do grafo da MEF em fig. 6 demostra como introduzir os estados adicionais (designados por "*dummy states*"). Estes permitem cortar algumas transições e realizá-las em mais que um ciclo de relógio (primeiro do estado inicial ao estado adicional e depois do estado adicional ao estado seguinte). Com base neste método é possível dividir qualquer transição em transições locais de tal maneira que cada uma destas é só causada por uma condição lógica. Como resultado, o circuito na fig. 4 pode ser usado. Mas este circuito vai funcionar mais devagar e não poderá ser utilizado nalgumas aplicações porque pode alterar as saídas activas no estado inicial e no estado seguinte (i.e. nos estados inicial e seguinte uma saída deve ser activa mas nos estados intermédios - "*dummy states*" a saída vai ser passiva e isso pode causar problemas). Este problema está resolvido no circuito da fig. 7.

| $a_{from}, Y(a_{from})$ | $K(a_{from})$ | $X(a_{from},a_{to})$ | $a_{to}$ | $K(a_{to})$ |
|---|---|---|---|---|
| $a_0$ (-) | 000 | 1 | $a_1$ | 001 |
| $a_1$ (-) | 001 | 1 | $a_1$ | 001 |
| $a_2$ ($y_1,y_2$) | 010 | $\bar{x}_1$ | $a_1$ | 001 |
| | | $x_1 \bar{x}_2$ | $a_4$ | 100 |
| | | $x_1x_2$ | $a_3$ | 011 |
| $a_3$ ($y_3$) | 011 | 1 | $a_4$ | 100 |
| $a_4$ ($y_4,y_5$) | 100 | $\bar{x}_1$ | $a_1$ | 001 |
| | | $x_1 \bar{x}_2$ | $a_4$ | 100 |
| | | $x_1x_2$ | $a_3$ | 011 |

Tabela 1 – Tabela das transições entre os estados da MEF (ver fig. 2)

| $a_{from}, Y(a_{from})$ | $K(a_{from})$ | $X(a_{from},a_{to})$ | $a_{to}$ | $K(a_{to})$ |
|---|---|---|---|---|
| $a_0$ (-) | 000 | 1 | $a_1$ | 001 |
| $a_1$ (-) | 001 | 1 | $a_1$ | 001 |
| $a_2$ ($y_1,y_2$) | 010 | $\bar{x}_1$ | $a_1$ | 001 |
| | | $x_1$ | $a_5^1$ | 101 |
| $a_3$ ($y_3$) | 011 | 1 | $a_4$ | 100 |
| $a_4$ ($y_4,y_5$) | 100 | $\bar{x}_1$ | $a_1$ | 001 |
| | | $x_1$ | $a_5^1$ | 101 |
| $a_5^1$ (-) | 101 | $\bar{x}_2$ | $a_4$ | 100 |
| (dummy state) | | $x_2$ | $a_3$ | 011 |

Tabela 2 – Tabela das transições com um estado $a_5^1$ adicional

Este circuito permite implementar qualquer transição que pode ser causada por uma ou duas condições lógicas. Consideremos o exemplo da transição do estado $a_2$ na tabela 1 para o estado $a_4$ que é influenciada por duas condições lógicas $x_1$ e $x_2$ incluídas na conjunção $x_1 \bar{x}_2$. Neste caso o código do estado inicial é $K(a_2)$=010 e este está guardado no registo Rg. Primeiro o circuito $STRAM_1$ gera o código intermédio $K(a_5^1)$=101 (ver tabela 2) que aparece nas saídas $T_1^1,...,T_R^1$ do circuito $STRAM_1$, onde o índice superior indica o nível do estado adicional. O código $K(a_5^1)$=101 não é guardado no registo Rg e é só considerado como um código intermédio no circuito combinatório (i.e. o código entre os níveis respectivos do circuito combinatório que é composto por vários PMs e STRAMs). A seguir, $STRAM_2$ converte o código $K(a_5^1)$=101 para o código $K(a_4)$=100 (ver tabela 2). Então a respectiva transição será realizada num ciclo de relógio. O registo Rg não vai guardar os códigos dos estados adicionais (i.e. *dummy states*) e estes só podem aparecer entre os níveis do circuito combinatório. Isto permite construir um circuito da MEFR que vai funcionar muito rapidamente. Vamos designar o conjunto formado do PM e STRAM do nível *i* como $LRAM_i$. Na fig. 7 temos $LRAM_1$ e $LRAM_2$ para dois níveis do circuito combinatório. Podemos considerar circuitos que possuam mais que dois níveis e isto permite implementar as transições que são causadas por mais que duas condições lógicas. Obviamente, se o circuito for composto por F níveis de LRAMs isto permite implementar as transições que são influenciadas por F condições lógicas.

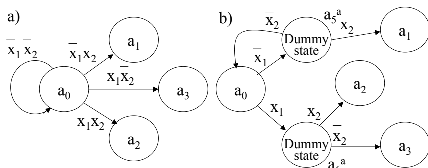


Fig. 6 - Um fragmento da MEF com estados adicionais (i.e. *dummy states*)

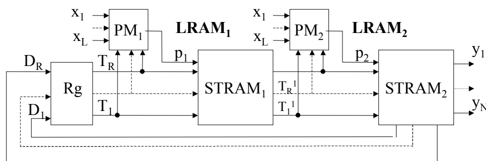


Fig. 7 - Segunda arquitectura da UCR

A modificação do comportamento da MEF é realizada através do re-carregamento dos blocos de memória (i.e. RAM). Existem várias FPGAs que incluem quer os blocos lógicos reconfiguráveis (CLB - *configurable logic blocks*) que podem ser utilizados como RAM simples (por exemplo, FPGA da família XC4000 da Xilinx [9]), quer uma memória distribuída (tal como a memória da FPGA Virtex XCV812E [10]). A MEFR com a arquitectura apresentada na fig. 7 pode ser construída para qualquer uma destas FPGAs.

## III. IMPLEMENTAÇÃO DA UNIDADE DE CONTOLO REPROGRAMÁVEL EM FPGA

Todos os circuitos para a FPGA foram desenvolvidos em *Xilinx Foundation Software* (versão 3.1). A fig. 8 mostra como o circuito da fig. 7 pode ser implementado com base na FPGA XC4010XL. Esta FPGA está incorporada na placa XS40 que faz parte da placa XStend [11]. O circuito é composto pelo registo RG3MEM (ver fig. 9) e duas LRAMs (blocos H5, H8 e blocos U3, H9). O bloco H10 é usado para realizar a comunicação com o computador através da porta paralela. Os circuitos de PM são implementados em blocos H5 e U3 para $LRAM_1$ e $LRAM_2$ respectivamente. O multiplexador M para os blocos PM (ver fig. 5) e todos os blocos da memória foram construídos com a ajuda de *LogiBLOX Module Generator* da Xilinx. O bloco U1 (IND_LED) estabelece as ligações com LEDs instalados na placa XStend. Isto permite verificar vários sinais duma forma visual.

Para implementar no circuito apresentado na fig. 8 o algoritmo da fig. 2 (ver também as tabelas 1 e 2) é necessário programar todos os blocos de memória, i.e. para cada bloco deve-se alterar o respectivo ficheiro *.mem da seguinte forma:

*$LRAM_1$ - $MRAM_1$*: 0:0, 1:0, 2:1, 3:0, 4:1, 5:2, 6:0, 7:0
*$LRAM_1$ - $STRAM_1$*: 0:2, 1:2, 2:1, 3:1, 4:1, 5:5, 6:4, 7:4, 8:1, 9:5, a:0, b:0, c:0, d:0, e:0, f:0
*$LRAM_2$ - $MRAM_2$*: 0:0, 1:0, 2:1, 3:0, 4:1, 5:2, 6:0, 7:0
*$LRAM_2$ - $STRAM_2$*: 0:0, 1:0, 2:1, 3:1, 4:2, 5:2, 6:3, 7:3, 8:4, 9:4, a:4, b:3, c:0, d:0, e:0, f:0

A MRAM permite seleccionar a entrada $x_1$ (i.e. $p=x_1$) nos estados $a_2$, $a_4$ com códigos $K(a_2)$=010, $K(a_4)$=100 (ver tabela 2) e a entrada $x_2$ (i.e. $p=x_2$) no estado adicional $a_5^1$ com código $K(a_5^1)$=101. Vamos verificar todas as transições do estado $a_2$ com código $K(a_2)$=010. Neste caso, consuante o valor da entrada $x_1$, $STRAM_1$ gera quer o código 001 (para $x_1$=0) quer o código 101 (para $x_1$=1). Os quatro bits do endereço da $STRAM_1$ são iguais ao $K(a_2)x_1$, i.e. são iguais quer a 0100 (para $x_1$=0) quer a 0101 (para $x_1$=1) e $STRAM_1$ contém no endereço 0100 o valor 001 (i.e. o código do estado $a_1$) e no endereço 0101 o valor 101 (i.e. o código do estado adicional $a_5^1$). Então a primeira transição do estado $a_2$ (i.e. $a_2\ \bar{x}_1 \Rightarrow a_1$) é uma transição directa e o código do estado $a_1$ é copiado das entradas do bloco $STRAM_2$ para as saídas do $STRAM_2$ e finalmente altera o estado do registo Rg (ver fig. 7). Agora vamos assumir que $STRAM_1$ gera o código do estado adicional $K(a_5^1)$=101. Neste caso, consuante o valor da entrada $x_2$, a $STRAM_2$ forma quer o código $K(a_4)$=100 (para $x_2$=0) quer o código $K(a_3)$=011 (para $x_2$=1). Então temos a transição $a_2 x_1\ \bar{x}_2 \Rightarrow a_4$ e a transição

$a_2x_1x_2 \Rightarrow a_3$, i.e. todas as transições do estado $a_2$ (ver tabela 1) são efectuadas correctamente.

Para implementar as saídas da MEFR é necessário incluir um bloco de memória que é mostrado na fig. 10. Este bloco, para o nosso exemplo, deve ser programado de acordo com o seguinte ficheiro *.*mem*: 0:0, 1:0, 2:3, 3:4, 4:18, 5:0, 6:0, 7:0. Por exemplo, a expressão 4:18 especifica o valor hexadecimal 18 (i.e. 11000 ou $y_4$ e $y_5$) que é escrito na memória no endereço 4 (i.e. 100). Como resultado, as microoperações $y_4$ e $y_5$ serão activadas no estado da MEF com o código 100 (i.e. no estado $a_4$).

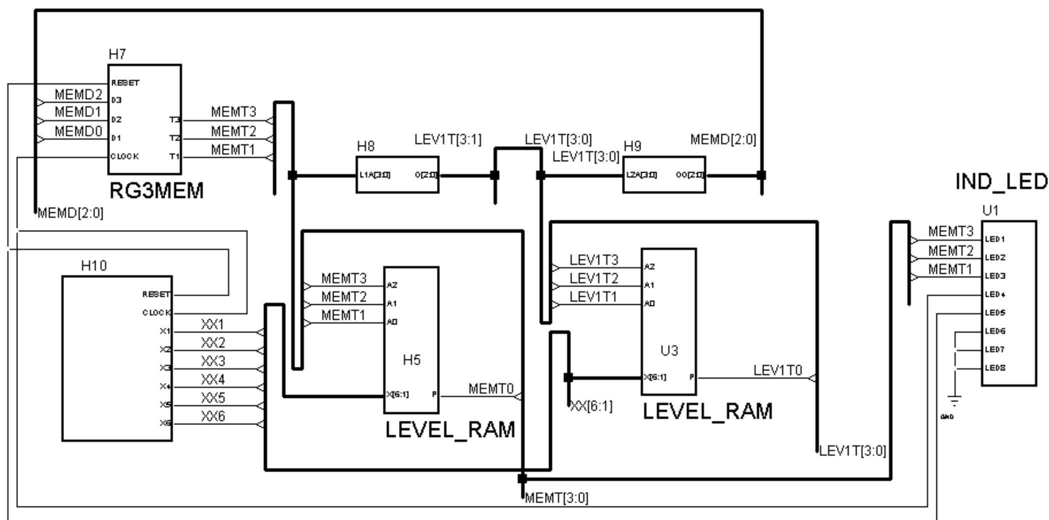


Fig. 8 - Implementação da MEFR com base na FPGA XC4010XL

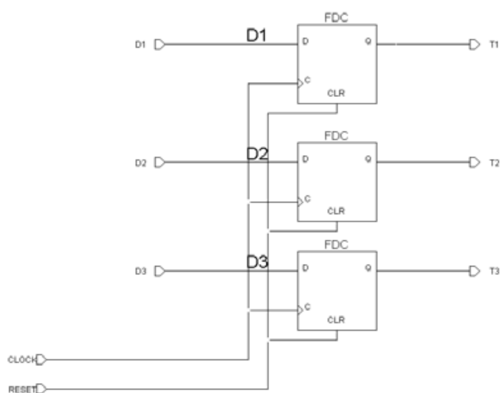


Fig 9 - Implementação do bloco REG3MEM (ver fig. 8)

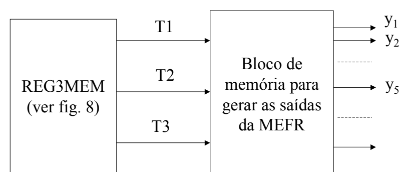


Fig 10 - Implementação da memória para saídas

## IV. SIMULAÇÃO DA MEFR EM SOFTWARE

Para representar o modelo do circuito mostrado na fig. 7; qual pode ter vários números de níveis de LRAM, foram criadas as três seguintes classes da linguagem C++:

```
class Register  //descrição do Rg (ver fig. 7)
{   public:
      void reset();  //reset do Rg
      unsigned T1TR(void)
               { return storage; }   // ler do Rg
      void D1DR(unsigned value)  // escrever no Rg
               { storage = value; }
                                   // inicializar o Rg
```

```
        Register(int r) : storage(0), R(r) {};
    // ... outras funções
      private:
        int R;                  // tamanho do Rg
        unsigned storage;   //memória para o Rg
    };

    class Prog_Mux  // descrição da MRAM (ver fig. 5)
    {   public:
    // Run devolve o valor "p" para o estado
    // e as entradas actuais
        unsigned Run(int state,unsigned X);
    // O construtor carrega os dados
    //  iniciais na MRAM
        Prog_Mux(int size,unsigned* data);
        virtual ~Prog_Mux();        // o destrutor

protected:
    int Size;        // o número de palavras na MRAM
    unsigned *RAM_M;  // a memória para MRAM
                                                };

    class Level_RAM  // descrição da LRAM (ver fig. 7)
    {   public:
// Run devolve o estado (real ou adicional)
// para o respectivo nível
        unsigned Run(int state,unsigned X);
    // init faz a initiacização da memória
    // para STRAM e MRAM
        void init(int R, unsigned* STRAM,
                    unsigned* MRAM);
        // o construtor da classe Level_RAM
        Level_RAM(int R, unsigned* STRAM,
                    unsigned* MRAM);
    // o construtor por defeito
    // para reservar arrays de LRAM
        Level_RAM() {};
        virtual ~Level_RAM();  // o destrutor
      private:
        Prog_Mux *PM;  // PM faz parte
                          // desta classe
        unsigned* storage;           // a memória
                                     // para STRAM
        int Size; // o número de palavras na STRAM
    };
```

Todas as funções das classes são muito simples. Por exemplo, a implementação da função *init* pode ser feita da seguinte forma:

```
  void Level_RAM::init(int R, unsigned *table,
                          unsigned *tableM)
  {  Size = (int)pow(2,R+1);  // é necessário
                             // incluir <math.h>
     PM = new Prog_Mux(Size/2,tableM);
     storage = new unsigned [Size];
     for(int i=0; i<Size; i++)
          storage[i] = table[i];                }
```

Agora podemos criar uma classe que descreva o circuito completo da MEFR (vamos chamar a esta classe **FSM_template**):

```
    class Register;          // declaração forward
    class Level_RAM;      // declaração forward
    class FSM_template
    {   public:
    // Run faz uma transição entre os estados reais
          unsigned Run(unsigned);
    // o construtor cria o circuito da MEFR
          FSM_template(int L, int R, int M, int N, int F,
            unsigned RAMI[][16]=0, // os dados
                                    // para STRAM
            unsigned *tableI=0,         // os dados
                                    // para MRAM
            unsigned *tableOUTI=0); // os dados
                                     // para saídas
          virtual ~FSM_template(); // o destrutor
          // Reload faz recarregamento
          // dos blocos de memória
    void Reload(unsigned RAM[][16], unsigned *table,
          unsigned *tableOUT);
    unsigned get_output(); // get_output devolve
                              // as saídas
    // outras funções
      private:
          int M_state;   // o número de estados
          int R_rg;         // o tamanho do registo
          int N_output; // o número de saídas
          int L_input;    // o número de entradas
          int Levels;      // o número de blocos LRAM
          unsigned* output_RAM;   // a memória
                                    // para saídas
          Level_RAM* array_LR; // a lista
                                  // das LRAMs
          Register FSMR;      }; // o registo da MEFR
```

A implementação das funções da classe **FSM_template** pode ser a seguinte:

```
    FSM_template::FSM_template(int L, int R, int M,
          int N, int F, unsigned RAMI[][16],
          unsigned *tableI,
          unsigned *tableOUTI)
          : FSMR(R), M_state(M), R_rg(R),
                N_output(N), L_input(L)
    {     array_LR = new Level_RAM[Levels=F];
          for (int i=0; i<Levels; i++)
            array_LR[i].init(R_rg,RAMI[i],tableI);
          output_RAM = new unsigned[M];
          for (i=0; i<M; i++)
            output_RAM[i] = tableOUTI[i];   }

    FSM_template::~FSM_template()
    {     delete []array_LR;
          delete []output_RAM;      }

    unsigned FSM_template::Run(unsigned int X)
```

```
{     unsigned propagate;
      propagate = FSMR.T1TR();
      for (int i=0; i<Levels; i++)
         propagate = array_LR[i].Run(propagate,X);
      FSMR.D1DR(propagate);
      return get_output();                    }

unsigned FSM_template::get_output()
{     return output_RAM[FSMR.T1TR()];   }

void FSM_template::Reload(unsigned RAM[][16],
      unsigned *table, unsigned *tableOUT)
{     delete []array_LR;
      delete []output_RAM;
      array_LR = new Level_RAM[Levels];
      for (int i=0; i<Levels; i++)
         array_LR[i].init(R_rg,RAM[i],table);
      output_RAM = new unsigned[M_state];
      for (i=0; i<M_state; i++)
         output_RAM[i] = tableOUT[i];
      FSMR.reset();                           }
```

## V. O MODELO EM C++ PARA A COMUNICAÇÃO ENTRE A MEFR E A UNIDADE DE EXECUÇÃO

Consideremos um circuito reprogramável que possa ser realizado como uma composição da UE e da MEFR (ver fig. 11). A UE é a mesma que está representada na fig. 3 com algumas entadas e saídas adicionais que são $x_3$, $y_6$ e $y_7$ i.e. $X=\{x_1,...x_3\}$, $Y=\{y_1,...,y_7\}$.

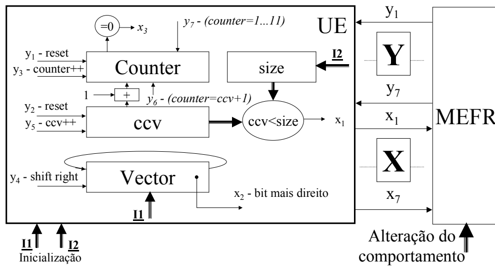


Fig. 11 - Exemplo do circuito reprogramável

O circuito na fig. 11 permite realizar várias operações sobre vectores binários tais como "contar o número de 1s num vector", "verificar se um vector só contém um 1 e encontrar a posição deste 1", etc. Por exemplo, para os vectores 010000 e 010110 a primeira operação dá os valores 1 e 3 e a segunda operação dá os valores 5 e F...FF onde F...FF (i.e. tudo a 1s) corresponde à resposta "o vector tem mais do que um 1".

Primeiro vamos verificar estas operações num programa desenvolvido em C++. Para estes fins introduzimos uma classe chamada **Boolean_vector**:

```
class Boolean_vector
{   public:
          short int number1s();    // primeira operação
          short int contain1one(); // segunda operação
          Boolean_vector(unsigned V=0, int S=0)
               : vector(V),size(S) {};    // o construtor
          virtual ~Boolean_vector(); // o destrutor
          // ...
    protected:
          void operator>>=(int); // redefinição de >>=
          short int size;       // ver fig. 11
          unsigned vector;  // ver fig. 11
    private:
          int counter;          // ver fig. 11
          int ccv;     };       // ver fig. 11
```

O código das funções *number1s*, *contain1one* está representado nas fig. 12 e fig. 13. De notar que estes códigos podem ser convertidos nos respectivos GSs (ver fig. 12 e fig. 13).

A primeira linha do código (i.e. *counter = 0*, ou realização da microoperação $y_1$ - ver fig. 11) é activada no nó $a_2$ (ver fig. 12). O ciclo *for(ccv=0; ccv<size; ccv++) {...}* inclui a inicialização do contador (i.e. *ccv=0*, ou $y_2$), a verificação da condição de terminação (i.e. *ccv<size*, ou $x_1$) e o incremento da variável (ccv) de controlo (i.e. *ccv++*, ou $y_5$). A linha **this >>= 1;* (ver microoperação $y_4$ nas fig. 11 e fig. 12) desloca o vector (i.e. *round shift right*). No programa será chamada a função *operador>>=* redefenida para a classe **Boolean_vector**. Esta função pode ter o seguinte código:

```
void Boolean_vector::operator>>=(int)
{       vector|=((vector&1)<<size);
        vector>>=1;                          }
```

A implementação em hardware é mostrada na fig. 11.

```
short int Boolean_vector::number1s()
{       counter=0;
        for(ccv=0; ccv<size; ccv++)
        {       if (vector & 1) counter++;
                *this >>= 1;                    }
        return counter;
}
```

| $a_{from}$ | $Y(a_{from})$ | $X(a_{from},a_{to})$ | $a_{to}$ |
|---|---|---|---|
| $a_0$ (000) | | 1 | $a_2$ (010) |
| $a_2$ (010) | $y_1,y_2$ | $\bar{x}_1$ | $a_1$ (001) |
| | | $x_1$ | $a_5^1$ (101) |
| $a_3$ (011) | $y_3$ | 1 | $a_4$ (100) |
| $a_4$ (100) | $y_4,y_5$ | $\bar{x}_1$ | $a_1$ (001) |
| | | $x_1$ | $a_5^1$ (101) |
| $a_5^1$ (101) | | $\bar{x}_2$ | $a_4$ (100) |
| | | $x_2$ | $a_3$ (011) |

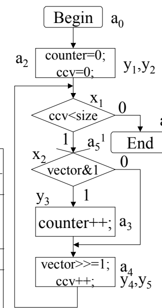


Fig. 12 - Implementação da operação *number1s*

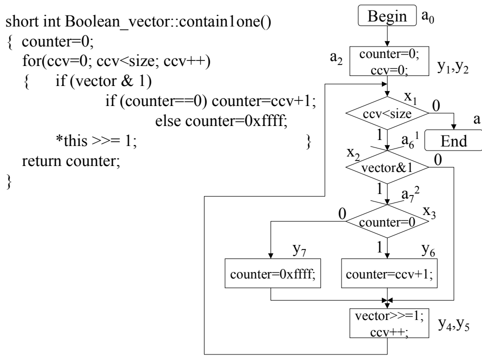


Fig. 13 - Implementação da operação *contain1one*

De facto o GS da fig. 12 é equivalente ao GS da fig. 2 e a tabela da fig. 12 é equivalente à tabela 2. Então todos os passos para implementar o respectivo algoritmo numa MEFR já foram tidos em consideração na secção II.

A fig. 13 mostra como fazer a mesma conversão da função *contain1one* no respectivo GS. A fig. 12 e a fig. 13 demonstram também como introduzir os estados adicionais para os vários níveis de LRAM (ver o estado $a_5^1$ na fig. 12 e os estados $a_6^1$ e $a_7^2$ na fig. 13).

Agora vamos simular em software todas as operações de hardware, i.e. vamos modelar a funcionalidade da UE e da MEFR (ver fig. 11). Para este fim introduzimos na classe **Boolean_vector** as funções específicas com nomes *Run* e *Solve*:

```
 unsigned Boolean_vector::Run(unsigned int Y)
{// implementação das microoperações (ver fig. 11)
        if (Y & 0x1) counter = 0;              // y1
        if (Y & 0x2) ccv=0;                    // y2
        if (Y & 0x4) counter++;                // y3
        if (Y & 0x8) *this >>= 1;              // y4
        if (Y & 0x10) ccv++;                   // y5
        if (Y & 0x20) counter=ccv+1;           // y6
        if (Y & 0x40) counter=0xffff;          // y7
// implementação das condições lógicas (ver fig. 11)
        unsigned X=0;
        if(ccv<size) X=X|0x1;                  // x1
        if(vector&1) X=X|0x2;                  // x2
        if(counter==0) X=X|0x4;                // x3
        return X;
}

unsigned Boolean_vector::Solve(FSM_template &fsm)
{          // "exchange" simula o barramento
           // entre UE e MEFR
        unsigned exchange=0;
        // 0x1000 é uma condição de terminação
        while (exchange!=0x1000)          {
                  // activação da UE
        exchange = Run(exchange);
                  // activação da MEFR
        exchange = fsm.Run(exchange);  }
        return counter;                         }
```

A função *Run* simula todas as operações da UE (ver fig. 11). A função *Solve* resolve o problema por activação da UE (ver a linha *exchange = Run(exchange);*) e da MEFR (ver a linha *exchange = fsm.Run(exchange);*), onde *fsm* é uma referência para a MEFR (ver a linha *Solve(FSM_template &fsm)*) que deve ser programada de acordo com qualquer um dos GSs nas fig. 12 e 13.

A reprogramação da máquina de estados finitos pode ser realizada das seguintes formas:

- com a ajuda duma função especial que se chama *Reload* (ver a classe **FSM_template** na secção II);
- por substituição de blocos de memória que realizam vários algoritmos do controlo.

A função *Reload* já foi abordada na secção II. A fig. 14 demonstra a ideia de substituição de blocos de memória.

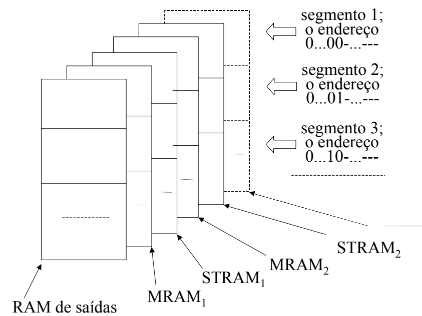


Fig. 14 – A substituição de blocos de memória

Cada bloco pode ser considerado como um segmento da mesma memória. Por exemplo, se a memória para implementação de várias funções da classe **Boolean_vector** tiver quatro blocos, cada bloco pode ter os seguintes endereços: *RAM de saídas* - 00--- (o bloco 0), 01--- (o bloco 1), 10--- (o bloco 2), 11--- (o bloco 3); *MRAM₁* - 00--- (o bloco 0), 01--- (o bloco 1), 10--- (o bloco 2), 11--- (o bloco 3); *STRAM₁* - 00---- (o bloco 0), 01---- (o bloco 1), 10---- (o bloco 2), 11---- (o bloco 3), etc., onde "-" significa qualquer valor, quer 0 quer 1.

Agora, a implementação em software da classe **FSM_template** pode ser feita de seguinte maneira.

```
class FSM_template
{  public:
      void reset(); // fazer reset da MEFR
      void change_segment(int); // trocar o segmento
      FSM_template(int L, int R, int M, int N, int G,
               int S, unsigned RAMI[][3][16]=0,
               unsigned tableI[][8]=0,
               unsigned tableOUTI[][8]=0);
```

```
        // ver o código na secção II
    private:            // a_seg é o número actual
        int a_seg;      // do bloco de memória
        // ver o código na secção II
};

void FSM_template::change_segment(int s)
{       a_seg = s;          }
```

A função *change_segment* troca o segmento actual *a_seg* para o segmento *s*. Em hardware esta operação pode ser realizada através da activação de bits que indicam o número do respectivo bloco (ver fig. 14). Para este fim podemos utilizar um registo adicional (ver fig. 15). Para o nosso exemplo em cima com quatro blocos de memória este registo pode conter 00 (para seleccionar o bloco 0), 01 (para seleccionar o bloco 1), 10 (para seleccionar o bloco 2) ou 11 (para seleccionar o bloco 3). A função *main* pode ser implementada da seguinte forma:

```
int main(int argc, char* argv[])
{           // especificação da MEFR
        FSM_template FSMT(3,3,8,7,3,2,
                tableRAM, table, Y);
          // especificação do vector
        Boolean_vector BV(0x2 ,5);
          // implementação do algoritmo da fig. 12
        cout << "number of 1s = "
                << BV.Solve(FSMT) << endl;
        FSMT.change_segment(1); // troca de blocos
          // implementação do algoritmo da fig. 13
        cout << "position of 1 = "
                << BV.Solve(FSMT) << endl;
        return 0;           }
```

O conteúdo dos blocos $B_1$ e $B_2$ para todos os níveis $L_1$, $L_2$, $L_3$ (S=2, G=3) do circuito combinatório da MEFR pode ser apresentado da seguinte forma:

*MRAMs*:

```
unsigned table[][8] =
    {{0x0,0x0,0x1,0x0,0x1,0x2,0x0,0x0},   // B1
     {0x0,0x0,0x1,0x0,0x1,0x0,0x2,0x3}};  // B2
```

*memória de saída*:

```
unsigned Y[][8] =
  {{0x0,0x1000,0x3,0x4,0x18,0x0,0x0,0x0},    // B1
   {0x0,0x1000,0x3,0x20,0x18,0x40,0x0,0x0}}; // B2
```

*STRAMs*:

```
unsigned tableRAM[][3][16] =
    {{{0x2,0x2,0x1,0x1,0x1,0x5,              // B1, L1
       0x4,0x4,0x1,0x5,0xf,0xf,0xf,0xf,0xf,0xf},
      {0x0,0x0,0x1,0x1,0x2,0x2,0x3,0x3,   // B1, L2
       0x4,0x4,0x4,0x3,0xf,0xf,0xf,0xf},
      {0x0,0x0,0x1,0x1,0x2,0x2,0x3,0x3, // B1, L3
       0x4,0x4,0x5,0x5,0x6,0x6,0x7,0x7}},
     {{0x2,0x2,0x1,0x1,0x1,0x6,0x4,0x4,      // B2, L1
       0x1,0x6,0x4,0x4,0xf,0xf,0xf,0xf},
      {0x0,0x0,0x1,0x1,0x2,0x2,0x3,0x3,   // B2, L2
       0x4,0x4,0x5,0x5,0x4,0x7,0xf,0xf},
      {0x0,0x0,0x1,0x1,0x2,0x2,0x3,0x3, // B2, L3
       0x4,0x4,0x5,0x5,0x6,0x6,0x5,0x3}}};
```

O código completo para o nosso exemplo em C++ está disponível na WebCT [12] para os alunos das disciplinas relevantes do DETUA. O projecto da MEFR para *Xilinx Foundation Software* está também disponível na WebCT [13].

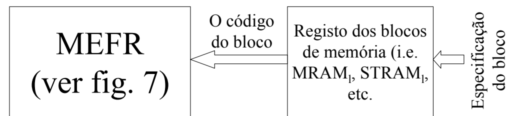


Fig. 15 - Troca de blocos de memória em hardware

## VI. SIMULAÇÃO EM C++ E DESENVOLVIMENTO DE CIRCUITOS DIGITAIS COM BASE EM HARDWARE TEMPLATES

Um hardware template (HT) é um circuito com uma arquitectura pre-definida e implementada em hardware (por exemplo, em FPGA). Vamos assumir que esta arquitectura pode servir de base para um conjunto de aplicações semelhantes tais como controladores de redes, problemas sobre matrizes binárias e ternárias, etc.

Se um HT pode ser utilizado para resolver vários problemas numa determinada área, devemos estar aptos a implementar algumas modificações no respectivo circuito. Vamos assumir que isto pode ser feito com base nos dois seguintes métodos:

1. Modificação dos algoritmos de controlo.
2. Extensão da arquitectura através da adição de alguns componentes.

A primeira possibilidade pode ser realizada com a ajuda da MEFR. Para implementar a segunda possibilidade podemos usar os métodos de programação orientada por objectos, i.e. a primeira arquitectura pre-definida é considerada como uma classe base e a extensão poderá ser feita numa classe derivada. Para hardware isto significa que o circuito com a arquitectura pre-definida faz parte dum circuito novo que inclui alguns componentes necessários para resolver um novo problema. A vantagem deste método é a reutilização de hardware. De notar que a MEFR descrita acima pode ser considerada como HT, mas deve-se especificar todos os parâmetros principais para construir a máquina de estados finitos, tais como: o número máximo $L_{max}$ de entradas, o tamanho máximo $R_{max}$ do registo, o número máximo $N_{max}$ de saídas, o número máximo $G_{max}$ de níveis, o número máximo $S_{max}$ de blocos reprogramáveis, etc. Por outras palavras é necessário construir a máquina quer em software quer em hardware. Em software este trabalho poder ser feito pelo construtor da classe (ver a função *main* na secção V). Em

software é necessário simular hardware reprogramável, tal como foi apresentado na fig. 8. Os parâmetros $L_{max}$ e $N_{max}$ especificam o barramento entre a UE e a UC (MEFR). A seguir podemos realizar a primeira possibilidade para alterar a funcionalidade do circuito com a ajuda da reprogramação dos blocos da MEFR. Isto permite implementar vários algoritmos de controlo que satisfazem as seguintes restrições $L \leq L_{max}$, $R \leq R_{max}$, $N \leq N_{max}$, $G \leq G_{max}$, $S \leq S_{max}$.

Um HT para a UE pode ser descrito em C++ da seguinte forma:

```
class datapath_template
{ public:
    datapath_template();   // o construtor
    virtual unsigned Run(unsigned Y)= 0;
    virtual unsigned Solve(FSM_template&)=0;
          // outras funções
    input Y;        // os tipos input e output descrevem
    output X;       // os vectores de entrada/saida
    // os componentes tais como
    // foram mostrados na fig. 11
          };
```

A classe **datapath_template** que descreve o HT é composta por funções de inicialização (tais como os construtores), por dados que representam vários componentes da UE (tais como registos, contadores, constantes, flip-flops individuais, etc.) e por funções para executar várias operações sobre os dados. Alguns dados e funções devem ser obrigatoriamente incluídos na classe. Estes são os seguintes:

- *input Y* que descreve as entradas da UE, i.e. os sinais gerados pela MEFR. O tipo *input* pode ser um tipo novo que permite descrever os vectores com vários tamanhos (i.e. com vários números de bits). Por outro lado este tipo pode ser, por exemplo, *unsigned*, o que para os computadores PC permite apresentar um vector com o tamanho de 32 bits;
- *output X* que descreve as saídas da UE, i.e. os sinais que influenciam o comportamento da MEFR. O tipo *output* pode ser um tipo novo que permite descrever os vectores com vários tamanhos (i.e. com vários números de bits). Este tipo pode ser também standard, por exemplo *unsigned*;
- as funções virtuais puras *Run* e *Solve*. Estas são virtuais e puras porque não podemos implementá-las sem definir o problema respectivo. A ideia principal destas funções é a mesma que a ideia das funções *Run* e *Solve* consideradas na secção V para a classe **Boolean_vector**. A função *Run* analiza vários bits de Y e executa as respectivas microoperações sobre os dados da classe **datapath_template** (porque cada microoperação está assocciada com o respectivo bit do vector Y). A seguir, a função *Run* devolve o vector X que mostra os estados de *flags* da UE (porque cada *flag* está assocciada com o respectivo bit do vector X). A função *Solve* tem como argumento uma referência para a MEFR que executa o respectivo algoritmo de controlo. A função *Solve* chama periodicamente as funções *Run* da MEFR e da UE até que seja obtida a solução do problema correspondente. A comunicação entre UE e MEFR é realizada através do barramento definido no corpo da função *Solve*.

Se a classe **datapath_template** for declarada como uma classe abstracta não será possível definir os objectos desta classe. Mas é permitido derivar uma classe nova da classe base **datapath_template** e esta classe nova será concreta, i.e. podemos definir os objectos desta classe. Consideremos um exemplo. Assume-se que a classe **datapath_template** é criada para resolver os problemas sobre as matrizes binárias e ternárias [6]. Vamos abordar um problema concreto tal como a determinação da cobertura de uma matriz (o artigo [6] define este problema em detalhe). Vamos criar uma classe nova derivada da classe **datapath_template** que se chama **covering**:

```
class covering : public datapath_template
{public:
   unsigned Run(unsigned Y);
   virtual unsigned Solve(FSM_template& fsm);
   virtual unsigned SolveP(); // a função adicional que
                  // resolve o problema só em software
   covering(int H_size,int V_size,
                  unsigned* matriz);  // o construtor
   virtual ~covering();                 // o destrutor
 private:
   void number1s();      // calcula o número de 1s
                         // num vector binário
   void set_masks(void); // indica as linhas e as
      // colunas da matriz que já foram encontradas
   void max_row(void); // para dada coluna encontra
                  // a linha que tem 1 na coluna
                  // e o valor máximo de 1s
   void transpose();     // transpõe a matriz
   void min_col(void); // encontra a coluna que tem
 };                    // o número mínimo de 1s
```

A função *SolveP* permite resolver o problema em software:

```
unsigned covering::SolveP()
{  unsigned Rg_const = (1<<h_size)-1;
             h_mask=0; v_mask=0;
   while (h_mask != Rg_const)   {
        min_col();
        if(Rg_E==0) return 0;
        max_row();
        set_masks();              }
        return v_mask;                    }
```

Registo *Rg_const* contém uma constante que tem valores 1s para colunas que devem ser cobertas. As variáveis *h_mask* e *v_mask* indicam as linhas e as colunas que já foram encontradas em vários passos do algoritmo

implementado na função *SolveP*. Este algoritmo foi descrito em [14] e inclui uma sequência de chamadas das funções *min_col*, *max_row* e *set_masks*. As funções *min_col* e *max_row* invocam a função *number1s* que calcula o número de 1s em colunas ou linhas. Todas estas funções são privadas porque só são chamadas no corpo da função *SolveP* que faz parte da classe **covering**.

Depois de verificar o problema em software podemos converter as funções *min_col*, *max_row*, *set_masks* e *number1s* aos respectivos algoritmos de controlo (i.e. aos respectivos GSs - ver secção V). Os GSs podem ser realizados numa MEFR. A referência para esta MEFR é passada como o argumento da função *Solve* da classe **covering**. Sendo assim, podemos simular e verificar todas as operações de hardware (ver secção V).

Agora vejamos como resolver o mesmo problema em hardware. Supomos que todos os componentes das classes **datapath_template** e **FSM_template** já são implementados em hardware. Se a classe **covering** não requerer os recursos adicionais devemos apenas modificar os ficheiros **.mem* para todos os blocos de memória. Existe só um problema. É necessário implementar as chamadas sequenciais e hierárquicas das nossas funções. De facto temos os dois níveis de chamada. As funções *min_col*, *max_row* e *set_masks* devem ser chamadas sequencialmente (nível 1) e a função *number1s* deve ser chamada nos corpos das funções *min_col* e *max_row* (nível dois). Estas chamadas podem ser realizadas com a ajuda da máquina de estados finitos hierárquica reprogramável (MEFHR) [2,15]. Cada função (tal como *min_col*, *max_row* etc.) pode ser considerada como um módulo. Para cada módulo devemos construir o respectivo GS e implementar estes GSs nos respectivos blocos (segmentes) de memória (ver fig. 14). O registo da MEFR tem que ser substituído por uma pilha (*stack memory*) que vai activar vários módulos duma forma hierárquica (ver fig. 16).

| | A pilha | | |
|---|---|---|---|
| Nível 2 | o bloco (ver fig. 14) | $a_0$ | Responsável pela função *number1s* |
| Nível 1 | o bloco (ver fig. 14) | $a_0$ | Responsável pelas funções *min_col*, *max_row* e *set_masks* |
| Nível 0 | o bloco (ver fig. 14) | $a_0$ | Responsável pela função *Solve* |

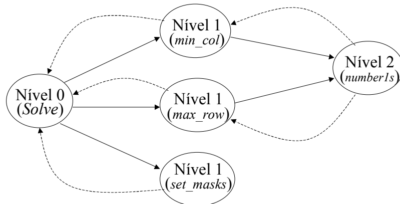


Fig. 16 – Chamada de vários módulos na MEFHR

Para o nosso exemplo a pilha deve conter pelo menos três registos (níveis 0, 1 e 2). No início o nível 0 é activado, i.e. a função *Solve* será invocada. Esta função tem o seguinte código:

```
unsigned covering::Solve(FSM_template& fsm)
{   unsigned exchange=0;
    while (exchange!=0x1000)      {
    exchange = Run(exchange);
    exchange = fsm.Run(exchange);
    if(exchange==m_c)  // m_c indica min_col
    // chamada da operação push sobre a pilha
    if(exchange==m_r)  // m_r indica max_row
   // chamada da operação push sobre a pilha
    if(exchange==s_m)  // s_m indica set_masks
   // chamada da operação push sobre a pilha
   if(exchange==n1s)  // n1s indica number1s
   // chamada da operação push sobre a pilha
   if(exchange==a1)  // a1 indica o estado a1
   // chamada da operação pop sobre a pilha
                                  }
    return v_mask;
  }
```

As variáveis m_c, m_r, s_m e n1s são sinais da MEFHR que correspondem às chamadas dos respectivos módulos: *min_col*, *max_row*, *set_masks* e *number1s*. Cada chamada é implementada através das seguintes operações:

- o bloco e o estado do módulo interrompido continuam no nível actual da pilha;
- o contador da pilha é incrementado (i.e. o nível novo é seleccionado);
- o bloco do módulo respectivo é escrito no nível novo e o estado inicial torna-se igual $a_0$ (ver fig. 16).

Como resultado o módulo de que precisamos será activado. Este módulo pode chamar o outro (ver na fig. 16 o grafo de chamadas possíveis para o nosso exemplo). Quando o módulo termina a sua execução, este entra num estado especial $a_1$. O sinal a1 para este estado chama a operação pop da pilha. Como resultado o controlo é retornado ao módulo interrompido no respectivo nível. Existem várias modificações da MEFHR [2]. Por exemplo, o estado inicial para cada módulo poder ser um estado que segue o estado $a_0$. Isto permite eliminar um ciclo de relógio para transição do estado $a_0$.

## VII. CONCLUSÕES

O artigo tem como o objectivo a apresentação de um material adicional para os alunos de disciplinas do DETUA tais como "Programação orientada por Objectos", "Sistemas Digitais Avançados", "Computação Reconfigurável", "Paradigmas de Programação", etc. O artigo demonstra a modelação em C++ e o desenvolvimento de sistemas digitais com base em hardware templates (i.e. os circuitos pré-construídos para uma área de aplicações semelhantes e reutilizáveis para vários problemas desta área). O código completo para a maioria dos programas descritos acima está disponível na WebCT [12,13,16] para os alunos daquelas disciplinas do DETUA. O projecto do modelo da máquina de estados

finitos reprogramável para *Xilinx Foundation Software* (versão 3.1) está também disponível na WebCT. A seguir está apresentada uma lista de artigos publicados em Português na revista de Electrónica e Telecomunicações que podem também ser úteis como materiais adicionais para os alunos das mesmas disciplinas.

- Os artigos [17,18] consideram os grafos de algoritmos hierárquicos e o modelo de máquina de estados finitos hierárquica.
- Os artigos [3,4] apresentam uma linguagem gráfica para descrição de vários algoritmos de controlo em sistemas digitais.
- O artigo [19] descreve um modelo de interligação de hardware (FPGA) e software. O software apresenta uma interface visual que permite mostrar todos os movimentos dum plotter. O hardware controla os movimentos e efectua as respectivas operações em software através da porta paralela.
- Os artigos [20,21,22] descrevem vários métodos de modelação e desenvolvimento de circuitos digitais com base nas FPGAs reconfiguráveis dinamicamente.
- O artigo [23] modifica a linguagem C++ para permitir simular algumas arquitecturas específicas tais como as utilizadas para os controladores CAN.
- O artigo [6] descreve vários problemas da área de optimização combinatória que podem ser formulados sobre matrizes binárias e ternárias.
- Os artigos [24,25] apresentam os resultados da implementação do processador MIPS com base na FPGA XC4010XL de Xilinx.
- Os artigos [26,27] descrevem um problema combinatório muito importante que se chama "Satisfação booleana". De notar que o artigo [27] contém uma lista bastante extensa da bibliografia nesta área, incluindo a utilização de vários recursos de computação reconfigurável.

## AGRADECIMENTOS

O autor agradece ao Prof. José Alberto Fonseca pela ajuda prestada na elaboração deste artigo.

## REFERÊNCIAS


[1] S.Baranov, "Logic Synthesis for Control Automata", Kluwer Academic Publishers, 1994.

[2] V.Sklyarov, Hierarchical Finite-State Machines and Their Use for Digital Control. IEEE Transactions on VLSI Systems, 1999, Vol. 7, No 2, pp. 222-228.

[3] A.Melo, V.Sklyarov, A.Ferrari, HiParaGraphs, uma Linguagem de Especificação de Algoritmos de Controlo Paralelos e Hierárquicos. Electrónica e Telecomunicações, Jan., Vol. 3, Nº 3, 2001, pp. 214-221.

[4] A.Melo, V.Sklyarov, A.Ferrari, Especificação e Simulação Interactiva de Algoritmos de Controlo Paralelos e Hierárquicos. Electrónica e Telecomunicações,, Sep., Vol. 3, Nº 4, 2001, pp. 332-344.

[5] A. Melo, Especificação, Optimização e Teste de Algoritmos de Controlo Hierárquicos, Tese de Mestrado, Aveiro, 2000.

[6] I.Skliarova, A.Ferrari, Modelos matemáticos e problemas de optimização combinatória. Electrónica e Telecomunicações, Jan., Vol. 3, Nº 3, 2001, pp. 202-208.

[7] V.Sklyarov, Synthesis and Implementation of RAM-based Finite State Machines in FPGAs. Proceedings of FPL'2000, Villach, Austria, August, 2000, pp. 718-728.

[8] V. Sklyarov, Applying Finite State Machine Theory and Object-Oriented Programming to the Logical Synthesis of Control Devices. Electrónica e Telecomunicações, 1996, Vol. 1, N 6, pp. 515-529.

[9] Xilinx, The programmable logic data book, Xilinx, San Jose, 2000.

[10] Virtex™-E 1.8 V Extended Memory Field Programmable Gate Arrays, 2000, www.xilinx.com.

[11] XStend board V1.2 Manual., XESS Corporation, 1998, 68 pages.

[12] Página *http://webct.ua.pt,* "1º Semestre", a disciplina "Programação por Objectos" com login alunoobj e password objectos (i.e. o acesso é o seguinte: http://webct.ua.pt/ ⇒ "Course Listing" ⇒ "1º Semestre" ⇒ "Programação Orientada por Objectos" ⇒ alunoobj ⇒ objectos).

[13] Página *http://webct.ua.pt,* "1º Semestre" a disciplina "Sistemas Digitais Avançados" com login alunocomp e password reconf (i.e. o acesso é o seguinte: http://webct.ua.pt/ ⇒ "Course Listing" ⇒ "1º Semestre" ⇒ " Computação Reconfigurável" ⇒ alunocomp ⇒ reconf).

[14] I.Skliarova, A.B.Ferrari, Design and Implementation of Reconfigurable Processor for Problems of Combinatorial Computations, Proceedings of the Euromicro Symposium on Digital System Design – DSD'2001, Warsaw, Poland, September 2001, pp.112-119.

[15] V. Sklyarov, Hierarchical Graph-Schemes, Latvian Academy of Science, Automatics and Computers, Riga, no 2, pp. 82-87, 1984

[16] Página *http://webct.ua.pt,* "2º Semestre", a disciplina "Paradigmas de Programação *I"* com login alunopp1 e password alunopp1 (i.e. o acesso é o seguinte: http://webct.ua.pt/ ⇒ "Course Listing" ⇒ "2º Semestre" ⇒ "Paradigmas de Programação" ⇒ alunopp1 ⇒ alunopp1).

[17] Valery Sklyarov, António Adrego da Rocha, Sintese de Unidades de Controlo Descritas por Grafos dum Esquema Hierarquicos, Electrónica e Telecomunicalções, vol. 1, no. 6, pp. 577-588, 1996.

[18] António Adrego da Rocha, Valery Sklyarov, Simulação em VHDL de Máquinas de Estados Finitas Hierárquicas. Electrónica e Telecomunicações, 1997, Vol. 2, N 1, pp. 83-94.

[19] V.Silva, F.Santos, V.Sklyarov, A interligação do Visual C++ com as FPGAs da XILINX. Electrónica e Telecomunicações, Jan., Vol. 2, Nº7, 2000, pp. 874-883.

[20] Valery Sklyarov, Andreia Melo, Arnaldo Oliveira, Nuno Lau, Ricardo Sal Monteiro, Circuitos Virtuais Baseados em Reprogramação e Reconfiguração Dinâmica. Electrónica e Telecomunicações, 1998, Vol. 2, N 2, pp. 248-260.

[21] A.Melo, V.Sklyarov, Ambiente Integrado para Especificação, Projecto e Verificação de Unidades de Controlo em FPGAs, Electrónica e Telecomunicações, Jan., Vol. 2, Nº4, 1999, pp. 477-485.

[22] A.Oliveira, V.Sklyarov, Especificação, Projecto e Implementação de Circuitos de Controlo Virtuais, Electrónica e Telecomunicações, Jan., Vol. 2, Nº4, 1999, pp. 487-495.

[23] A.Oliveira, V.Sklyarov, A.Ferrari, EaSys - Uma Linguagem Orientada por Objectos para Descrição de Sistemas Digitais. Electrónica e Telecomunicações, Sep., Vol. 3, Nº 4, 2001, pp. 311-331.

[24] I.Skliarova, A.B.Ferrari, Implementação e Simulação do Processador MIPS com a ALU reconfigurável dinamicamente,

Electrónica e Telecomunicações, v.2, Nº3, January 1999, pp. 497-504.

[25] I.Skliarova, A.B.Ferrari, Projecto e Implementação de um subconjunto da arquitectura MIPS16 com base em FPGA XC4010XL, Electrónica e Telecomunicações, v.2, Nº6, September 1999, pp. 724-732.

[26] I.Skliarova, A.B.Ferrari, Utilização de hardware reconfigurável para acelerar a satisfação booleana, Electrónica e Telecomunicações, September 2001, pp. 304-310.

[27] I.Skliarova, A.B.Ferrari, Satisfação booleana: algoritmos, aplicações, implementações, Electrónica e Telecomunicações, 2002 (nesta revista).