Henrique Dezani

Geração Automática de Código para Microcontroladores Aplicada a um Ambiente de Co-projeto de *Hardware* e *Software*

Orientador:
Prof. Dr. Norian Marranghello

Mestrado em Engenharia Elétrica
Departamento de Engenharia Elétrica - DEE
Faculdade de Engenharia de Ilha Solteira - UNESP/FEIS

Ilha Solteira – SP

Maio / 2006

Henrique Dezani

Geração Automática de Código para Microcontroladores Aplicada a um Ambiente de Co-projeto de *Hardware* e *Software*

Dissertação apresentada à Faculdade de Engenharia de Ilha Solteira da UNESP de Ilha Solteira para obtenção do título de Mestre em Engenharia Elétrica

Orientador:
Prof. Dr. Norian Marranghello

Mestrado em Engenharia Elétrica
Departamento de Engenharia Elétrica - DEE
Faculdade de Engenharia de Ilha Solteira - UNESP/FEIS

Ilha Solteira – SP

Maio / 2006

Dedico este trabalho aos meus pais, Rui Dezani e Lucianete Mendes Dezani, ao meu irmão Eduardo Dezani e a uma mulher muito especial em minha vida, Gisele de Fátima Galan.

AGRADECIMENTOS

Ao meu orientador, professor e amigo Norian Marranghello, pelo constante incentivo, sempre indicando a direção a ser tomada nos momentos de maior dificuldade. Agradeço, principalmente, pela confiança depositada em mim e no meu trabalho de dissertação.

A professora Dra. Rogéria Cristiane Gratão de Souza, por sua participação na minha banca de qualificação e defesa do mestrado e, especialmente, por suas sugestões e ajudas dadas para a melhoria deste trabalho.

Ao professor Dr. Carlos Magnus Carlson Filho, pela sua participação na minha banca de defesa do mestrado e, principalmente, pelo apoio que me deu durante todos estes anos de minha vida acadêmica.

Ao professor Dr. Alexandre César Rodrigues da Silva, pela participação na minha banca de qualificação e pelas sugestões dadas ao meu trabalho.

Ao professor Dr. Aledir Silveira Pereira, pela sua participação em minhas bancas de estudos especiais I e II e pelas sugestões dadas ao meu trabalho.

A todos os professores, funcionários e amigos do Departamento de Engenharia Elétrica da Faculdade de Engenharia de Ilha Solteira/UNESP, em especial ao professor Dr. Josué Vieira Filho e aos meus amigos Marcelo Luis Murari e Giorjety Licorini Dias.

Ao Raimundo Barreto e Meuse Oliveira Junior, do Centro de Informática da Universidade Federal de Pernambuco, pela ajuda que me deram no início do meu trabalho.

As professoras M.Sc. Edna Yoshiko Senzako e Dra. Valéria Maria Volpe, pela ajuda que me deram durante os meus estudos e pela paciência com que responderam a todos os meus e-mails.

Aos meus pais Rui Dezani e Lucianete Mendes Dezani e ao meu irmão Eduardo Dezani, pelo estímulo e apoio incondicional, pela paciência e grande amizade com que sempre me ouviram, e sensatez com que sempre me ajudaram.

E a Gisele de Fátima Galan, uma mulher incrível que sempre me ajudou durante todo o mestrado, corrigindo meus textos e, principalmente, ficando ao meu lado e sempre me apoiando durante estes anos com muito amor.

RESUMO

Neste trabalho descreve-se um programa de geração automática de código para o microcontrolador 8051 da Intel, a partir de uma rede de Petri, com o objetivo de minimizar o tempo gasto na codificação do programa e automatizar completamente este processo de transformação. Definiu-se o uso da rede de Petri Lugar/Transição como modelo de entrada pois, mesmo tendo um modelo mais compacto, a rede de Petri Colorida, quando transformada em código *Assembly* é consideravelmente maior que o código *Assembly* gerado para a rede de Petri Lugar/Transição. Conclui-se que o código gerado pelo programa corresponde, exatamente, ao modelo da rede e pode ser executado pela arquitetura-alvo sem a necessidade de alterações no código.

Palavras-chave: Co-projeto de *hardware* e *software*, Síntese de *Software*, Microcontroladores, Redes de Petri.

ABSTRACT

This dissertation describes a program for the automatic generation of microcontroller code. The program takes a Petri net as input and outputs the corresponding assembly code for the Intel's 8051. The goal of this work is to speed up the coding process as well as to completely automate such a transformation. We use place/transition nets because even colored Petri nets resulting in quite compact models the assembly codes produced from them are much larger than those produced from place/transition nets. Also the code generated by the program described here exactly matches the net model, and can be directly executed on the target architecture without the need for further tuning.

Keywords: Hardware/Software Co-design, Software Synthesis, Microcontrollers, Petri Net

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SIGLAS

UCP	Unidade Central de Processamento
HDL	*Hardware Description Language*
PC	*Program Counter*
IR	*Instruction Register*
ROM	*Read Only Memory*
RAM	*Random Access Memory*
PSEN	*Program Store Enable*
EA	*External Access*
PSW	*Program Status Word*
DPTR	*Data Pointer*
ALE	*Address Latch Enable*
UML	*Unified Modeling Language*
PNML	*Petri Net Markup Language*
XML	*Extensible Markup Language*
SML/NJ	*Standard ML of New Jersey*
JVM	*Java Virtual Machine*

INTRODUÇÃO

Com o passar dos anos a indústria eletrônica teve um grande crescimento de vendas devido ao uso de componentes eletrônicos em diversos sistemas, tais como automóveis, eletrodomésticos e dispositivos de comunicação [De Micheli e Sami 1996].

Estes componentes estão se tornando cada vez mais complexos e com um tempo disponível para sua colocação no mercado cada vez menor. Por isso, surgiu a necessidade da criação de um ambiente de projeto mais eficiente, no qual o projetista pudesse informar o que desejava que o sistema realizasse e que lhe retornasse um protótipo. Fala-se sobre os ambientes de co-projeto de *hardware* e *software*, os quais têm sido muito usados no projeto de sistemas embutidos [Wolf 2003]. Estes ambientes são compostos por várias ferramentas que auxiliam o projetista nas diversas etapas do projeto. As principais etapas são o particionamento, a co-simulação e a co-síntese, a qual pode ser dividida em síntese de *hardware* e síntese de *software*.

Com o objetivo de reduzir o tempo gasto na síntese de *software*, criou-se uma ferramenta para a geração automática de código para sistemas embutidos a partir de um modelo formal descrito em Rede de Petri Lugar/Transição. Esta ferramenta gera um código na linguagem de montagem do microcontrolador 8051 da Intel [Intel 1994], o qual foi escolhido por ser uma arquitetura muito usada em sistemas de controle de processos.

Nesta dissertação é descrita a estrutura e a funcionalidade desta ferramenta, assim como os métodos que a constituem e os testes realizados com diversas redes de Petri Lugar/Transição.

1 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo são descritos os recursos utilizados pelo sistema de geração automática de código, tais como o ambiente de co-projeto para o qual será aplicado, a arquitetura-alvo e a linguagem na qual será gerado o código e a rede de Petri de entrada.

1.1 Ambiente de Co-projeto de *Hardware* e *Software*

Ambientes de co-projeto de *hardware* e *software* são utilizados para desenvolver o projeto de sistemas digitais, principalmente sistemas de propósitos específicos como os sistemas embutidos, ou embarcados.

Nestes ambientes, o projetista entra com uma especificação do sistema que deseja obter, onde informa tanto os requisitos funcionais quanto os requisitos não-funcionais. Em seguida, esta especificação é analisada e sintetizada de forma automática pelas etapas do ambiente até que seja gerado um protótipo do sistema já testado e simulado. Existem diversas estruturas desses ambientes, as quais diferenciam-se por suas etapas [De Michelli e Sami 1996], [Niemann 1998] e [Wolf e Staunstrup 1997].

Por meio destes ambientes, pretende-se agilizar o processo de desenvolvimento dos sistemas, diminuindo o tempo para colocação de produtos no mercado e tendo todas as informações referentes ao tempo de desenvolvimento e custo controlados.

A estrutura do ambiente de co-projeto utilizado para a realização deste trabalho é mostrada na Figura 1 e foi adaptada do modelo de Niemann [Niemann 1998]. Este modelo segue a seguinte seqüência de etapas: especificação, particionamento, co-síntese, co-simulação e validação, as quais serão descritas no decorrer desta seção.

A primeira etapa do modelo consiste na especificação do sistema por meio de um modelo formal, tal como redes de Petri, máquinas de estados finitos, grafos de fluxo de dados

e linguagens de programação [Niemann 1998]. Neste trabalho, serão utilizadas as redes de Petri, as quais serão discutidas em detalhes na seção 1.3 deste capítulo. Na especificação do sistema, o projetista deve informar, além de sua funcionalidade, seus requisitos não-funcionais, tais como: custo, desempenho esperado, tamanho físico dos componentes, entre outros itens relevantes à caracterização do sistema.

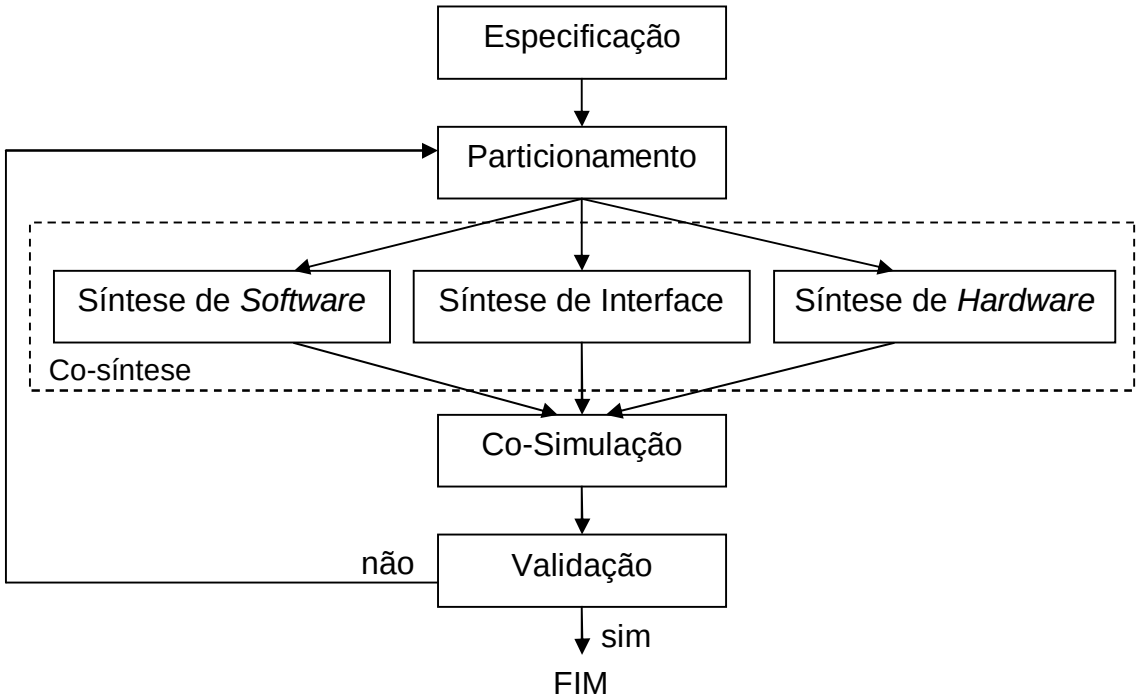


Figura 1. Ambiente de co-projeto utilizado neste trabalho, adaptado de [Niemann 1998].

A segunda etapa, chamada de particionamento, recebe a especificação do sistema e a divide em módulos menores que contêm a descrição de cada componente. A decisão de quais componentes serão implementados em *hardware* e quais serão implementados em *software* é feita a partir de estimativas de custo e desempenho dos componentes de *hardware* e *software.* Para *hardware* considera-se tempo de execução, área ocupada pelo componente, consumo de energia. Para *software* considera-se tempo de execução, quantidade de memória necessária para dados e programa. Os custos são equivalentes aos valores das partes físicas do equipamento e do tempo para desenvolvimento; portanto, quanto maior e mais complexo for um componente mais caro ele será. Já o desempenho é calculado por meio do tempo de execução das tarefas que o componente deve realizar.

Em geral, as partes que precisam de um maior desempenho são implementadas em *hardware*, o que aumenta o custo final do sistema, por causa da utilização de mais componentes físicos. Por outro lado, quando o custo é mais relevante que o desempenho, o sistema é implementado em *software*, fazendo tanto o custo como o desempenho diminuírem [De Michelli e Sami 1996].

A etapa de co-síntese recebe os módulos criados pela etapa de particionamento e realiza a síntese de alto nível destes módulos. Nesta etapa são executadas três funções concorrentemente: síntese de *hardware*, síntese de interface e síntese de *software*.

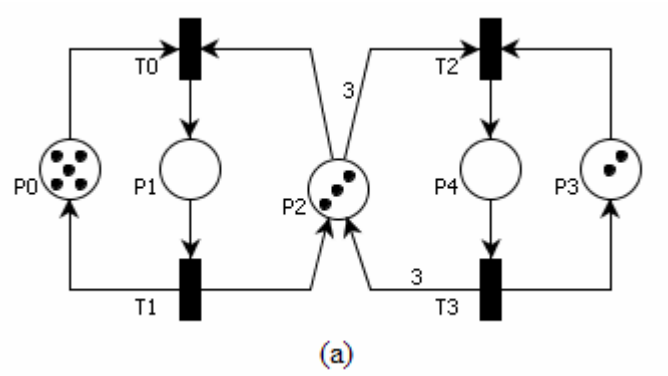
Por meio da síntese de *hardware* é possível realizar a conversão dos módulos de *hardware*, criados na etapa de particionamento, em uma linguagem de descrição de *hardware* (HDL – *Hardware Description Language*), tais como VHDL e Verilog-HDL.

A síntese de interface resulta em programas de acionamento (*driver*) e circuitos lógicos que provêm conexões físicas entre um processador e os dispositivos com os quais ele deve se comunicar. Esta comunicação pode ser realizada de duas maneiras, quais sejam, alocação de portas físicas do processador para os dispositivos ou a seleção de uma rotina de *software* que controle os acessos dos dispositivos ao processador. [Gupta 1993]

A síntese de *software*, na qual se insere a ferramenta proposta neste trabalho, possibilita a conversão dos módulos de *software*, resultantes da etapa de particionamento, em códigos para uma determinada linguagem de programação, os quais serão posteriormente compilados para a arquitetura-alvo, definida na etapa de particionamento, tal como um microcontrolador ou um microprocessador.

Tanto na síntese de *software* como na síntese de *hardware*, os módulos são transformados para linguagens que descrevem o sistema e não são propriamente implementados em componentes de *hardware* ou *software*. Para isso é necessário que os códigos gerados sejam compilados.

Na Figura 2 tem-se um exemplo de um modelo de *software* feito em rede de Petri Lugar/Transição que representa um sistema de escrita e leitura, o qual será detalhado no último capítulo, e seus respectivos códigos PNML (*Petri Net Markup Language*), gerado a partir da ferramenta PIPE [PIPE 2005], e *Assembly*, gerado pela ferramenta de síntese de *software*. A etapa de síntese de *software* e, conseqüentemente, o programa, recebe o modelo na forma de código PNML e retorna o código *Assembly* para a arquitetura-alvo desejada.



(a)

```
<pnml>
    <place id="P4">
        <name>
           <value>P4</value>
        </name>
        <initialMarking>
           <value>0</value>
        </initialMarking>
    </place>

    ...

    <transition id="T3" />

    ...

    <arc id="P4 to T3"
       source="P4" target="T3">
        <inscription>
           <value>1</value>
        </inscription>
    </arc>
    ...
</pnml>
```

(b)

```
MOV 15h,#5h
MOV 17h,#2h
MOV 18h,#3h

ET3I:  MOV A,16h
       CLR C
       SUBB A,#1h
       JC ET3F
       DEC 16h
       INC 17h
       MOV A,18h
       ADD A,#3h
       MOV 18h,A
ET3F:  LJMP ET2I
...
       END
```

(c)

Figura 2. Processo de síntese de *software*: (a) Modelo de rede de Petri (b) Código PNML referente ao modelo da rede de Petri (c) Código *assembly* gerado a partir do código PNML

A penúltima etapa deste ambiente consiste na simulação dos componentes de *hardware* sintetizados junto ao processador, executando os códigos. Esta etapa é chamada de

co-simulação e retorna as estimativas de desempenho e custo dos sistemas, explicadas anteriormente. Assim, a última etapa, chamada de validação, verifica os resultados gerados pela co-simulação e, se satisfizerem as metas estabelecidas na especificação do sistema, o ambiente de co-projeto é finalizado, retornando como saída um protótipo testado e validado, pronto para ser implementado. Porém, se tais metas não forem satisfeitas, o ambiente de co-projeto retorna à etapa de particionamento, escolhendo outra alternativa para a implementação dos componentes em *hardware* ou em *software* [Gupta 1993].

1.2 Sistemas Embutidos

Existem duas classes de sistemas digitais, quais sejam, sistemas de propósito geral e sistemas de propósito específico. Os sistemas de propósitos gerais, tais como os computadores pessoais e os supercomputadores, suportam uma grande variedade de aplicações, definidas por programas, que atendem a diversas tarefas. Por outro lado, os sistemas de propósitos específicos possuem sua funcionalidade bem definida e são usados dentro de um sistema maior, onde realizam funções dedicadas [Niemann 1998].

Os sistemas embutidos são responsáveis por mais da metade do mercado mundial de sistemas de processamento eletrônico e estão presentes no cotidiano das pessoas na maioria dos produtos, tais como eletrodomésticos, sistemas de telefonia móvel e sistemas automotivos. Estes sistemas, em geral, reagem a eventos externos do sistema em tempo real, por isso são considerados sistemas reativos, ou seja, sistemas que respondem a estímulos externos ao sistema, mudando seu estado e produzindo um novo resultado em um espaço de tempo pré-definido. Além disso, estes sistemas podem requerer uma carga maior de processamento de dados (sistemas de processamento de dados embutidos) ou uma carga maior nas atividades de controle (sistemas de controle embutidos), que não necessitam de grandes quantidades de espaço para armazenamento e possuem um desempenho de nível baixo a médio [Niemann 1998] e [Gimenez 2002].

O foco deste trabalho são os sistemas de controle embutidos, os quais são freqüentemente implementados em microcontroladores. Devido a outras atividades dentro do grupo de pesquisa, nos quais se utiliza o microcontrolador 8051 da Intel como unidade de processamento, o desenvolvimento da ferramenta está centrado nesta arquitetura, a qual será descrita na próxima seção.

1.2.1 Microcontrolador 8051 da Intel

O microcontrolador 8051 da Intel é composto por uma UCP de 8 *bits* otimizada para aplicações de controle, 128 *bytes* de memória de dados interna e capacidade para endereçar até 64 *Kbytes* de memória de dados externa, 4 *Kbytes* de memória de programa interna e capacidade de endereçar até 64 *Kbytes* de memória de programa externa, quatro portas de entrada e saída, dois temporizadores/contadores de 16 *bits*, uma interface serial, processamento booleano e cinco entradas de interrupção [Gimenez 2002]. Na Figura 3 é mostrado o diagrama em blocos da arquitetura interna do microcontrolador 8051.

O funcionamento do microcontrolador é iniciado ao receber um sinal de nível lógico 1 em seu pino de controle *reset*, o qual é responsável por iniciar o registrador interno da UCP, chamado de contador de programa (PC – *Program Counter*). Este registrador contém o endereço da próxima instrução de programa a ser buscada na memória de programa e executada pela UCP. Após a UCP realizar a busca da instrução, ele a armazena em um registrador de instrução (IR – *Instruction Register*), o qual contém a instrução atual que está sendo executada pela UCP, e armazena em PC o endereço da próxima instrução a ser executada.

Como se vê na Figura 3, o microcontrolador 8051 da Intel contém memória ROM (*Read Only Memory*) e memória RAM (*Random Access Memory*). Na memória ROM é armazenado o programa que será executado pela UCP do microcontrolador, também chamado de *firmware*.

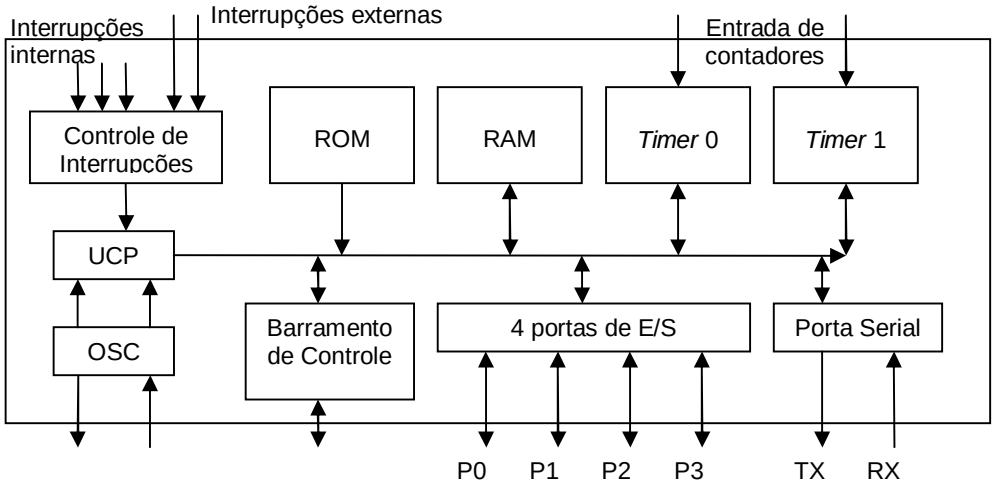


Figura 3. Arquitetura do microcontrolador 8051 da Intel

O microcontrolador 8051 possui capacidade de endereçar até 64 *Kbytes* de memória de programa, dos quais 4 *Kbytes* podem ser utilizados como memória interna ou os 64 *Kbytes* podem endereçar apenas a memória ROM externa, como se pode ver na Figura 4. Para alterar entre o uso da memória de programa interna ou externa é necessário utilizar dois dos quatro pinos de controle do microcontrolador, quais sejam, PSEN\ e EA\ [Nicolosi 2000].

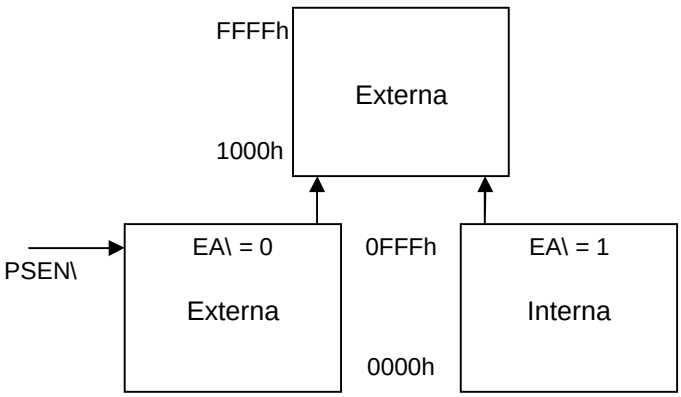


Figura 4. Organização da memória de programa do microcontrolador 8051 da Intel

O pino PSEN\ (*Program Store Enable*) é responsável por acionar a memória de programa externa e trabalha com nível lógico baixo. O pino EA\ (*External Access*) é responsável por determinar o uso da memória ROM interna do *chip* ou apenas a memória ROM externa e também trabalha com nível lógico baixo. Portanto, quando o pino EA\ estiver em nível lógico 1, o microcontrolador lerá a ROM interna, e após acabar todo o espaço da

memória interna, automaticamente, lerá a ROM externa a partir do endereço 1000h, se o pino PSEN\ estiver em nível lógico 0. Caso contrário, se o pino EA\ estiver em nível lógico 0, o microcontrolador utilizará apenas a ROM externa, caso o pino PSEN\ esteja em nível lógico 0, ou seja, caso a memória ROM externa esteja acionada.

Os pinos que controlam o acesso à memória de dados externa são WR e RD. Estes dois pinos são referentes aos pinos 6 e 7, respectivamente, da porta P3. Além da ligação física do microcontrolador com o *chip* de memória de dados externa, é necessário utilizar instruções diferentes para acessar cada uma dessas memórias. Para utilizar a memória de dados externa, usa-se a instrução MOVX (MOV eXtern) e para utilizar a memória de dados interna, utiliza-se a instrução MOV [Gimenez 2002]. Estas instruções e os modos de endereçamento são discutidos nas próximas seções.

A memória de dados interna é dividida em três partes. A primeira parte é constituída por quatro bancos de oito registradores de 8 *bits*, que ocupam os endereços de 00h a 1Fh (32 *bytes*). Na programação do microcontrolador, estes registradores são referenciados como R0, R1, R2, R3, R4, R5, R6 e R7. Apenas um dos quatro bancos de registradores pode ser acessado por vez e sua seleção é feita através de dois *bits* do registrador de função especial PSW (*Program Status Word*) chamados RS0 e RS1. A Tabela 1 mostra a configuração dos registradores para a seleção dos bancos.

Tabela 1. Configuração dos registradores para a seleção do banco

RS1	RS0	Banco selecionado	Endereços de memória selecionados
0	0	0	00h a 07h
0	1	1	08h a 0Fh
1	0	2	10h a 17h
1	1	3	18h a 1Fh

A segunda parte da memória é utilizada para propósitos gerais e ocupa os endereços de 20h a 2Fh (16 *bytes*). Esta parte da memória pode ser endereçada por *bytes* ou por *bits*, a qual, quando endereçada por *bits*, vai do endereço 00h a 7Fh. Por fim, a terceira parte da memória também é utilizada para propósitos gerais e vai do endereço 30h a 7Fh [Gimenez 2002].

O microcontrolador possui também 128 *bytes* nos quais são armazenados os registradores de funções especiais. Estes registradores são utilizados para controlar o microcontrolador, como por exemplo, selecionar o banco de registrador a ser usado na memória de dados e habilitar uma das portas P0, P2 e P3 para a comunicação com memórias externas. Na Tabela 2 são mostrados todos os registradores de funções especiais e em seguida, no decorrer do texto, serão explicados os mais importantes para o trabalho, que são o Acumulador (A), o *Data Pointer* (DPTR), a *Program Status Word* (PSW) e as portas P0 a P3. Os registradores marcados com asterisco podem ser endereçados tanto por *bytes* como por *bits*.

O registrador A ou ACC é utilizado em várias instruções do microcontrolador e, em geral, é parte do operando e, após as operações lógicas ou aritméticas, armazena o resultado. Os registradores DPH (*Data Pointer High*) e DPL (*Data Pointer Low*) juntos formam um registrador de 16 *bits* chamado DPTR, o qual é muito utilizado para armazenar o endereço de memória externa.

Tabela 2. Registradores de funções especiais

Registradores	Endereços	Nome dos registradores
A ou ACC	E0h	Acumulador
B*	F0h	Registrador B
DPL	82h	*Byte* menos significativo do ponteiro de dados (DPTR)
DPH	83h	*Byte* mais significativo do ponteiro de dados (DPTR)
IE*	A8h	Habilitador de interrupções
IP	B8h	Priorizador de interrupções
SCON*	98h	Controlador de comunicação serial
SBUF	99h	*Buffer* de dados serial
PSW*	D0h	Palavra de status do programa
PCON	87h	Controle de potência
TMOD	89h	Modo de operação do temporizador/contador
TH0	8Ch	*Byte* mais significativo do temporizador/contador 0
TL0	8Ah	*Byte* menos significativo do temporizador/contador 0
TH1	8Dh	*Byte* mais significativo do temporizador/contador 1
TL1	8Bh	*Byte* menos significativo do temporizador/contador 1
P0*	80h	Porta 0
P1*	90h	Porta 1
P2*	A0h	Porta 2
P3*	B0h	Porta 3

O registrador PSW é utilizado para armazenar o estado da última operação lógica ou aritmética realizada no acumulador, e com isso é possível manipular o programa para executar

uma instrução ou outra de acordo com o seu resultado. Na Figura 5 tem-se os *bits* deste registrador e em seguida a explicação de cada um deles [Gimenez 2002] e [Nicolosi 2000].

	Bit 7	*Bit* 6	*Bit* 5	*Bit* 4	*Bit* 3	*Bit* 2	*Bit* 1	*Bit* 0
PSW =	C	AC	F0	RS1	RS0	OV		P

Figura 5. Registrador de função especial PSW

O *bit* 7 é chamado de *Carry Bit* (C) e é utilizado para indicar o "vai um" em operações aritméticas de adição e subtração. Após as operações lógicas, esse *bit* é zerado. Além disso, ele pode ser utilizado para comparação entre dois valores após uma operação de subtração. Se o valor do *Carry Bit* for zero, significa que o primeiro operando é maior ou igual ao segundo operando, caso contrário, se o valor do *Carry Bit* for um, significa que o primeiro operando é menor que o segundo. Esta função de comparação será muito utilizada no programa-alvo.

O *bit* 6 é chamado de *Auxiliary Carry Bit* (AC) e é utilizado para indicar o "vai um" em operações aritméticas de adição e subtração do *bit* 3 do valor do resultado. Após operações lógicas este *bit* é configurado para nível lógico 1 (AC = 1).

O *bit* 5 é chamado de *General Purpose Flag* (F0) e é utilizado para propósito geral, podendo ser zerado ou configurado para o nível lógico 1 por *software* para indicar alguma condição desejada.

Os *bits* 3 e 4 (RS0 e RS1) foram citados anteriormente e são utilizados para selecionar um dos quatro bancos registradores da memória de dados.

O *bit* 2 é chamado de *Overflow Flag* (OV) e é utilizado para representar um número sinalizado. Quando está em nível lógico 0, o *bit* mais significativo representa o sinal (0 = positivo e 1 = negativo) e os outros 7 *bits* restantes representam o valor.

O *bit* 0 é o último deste conjunto e é chamado de *Parity flag* (P). Ele é utilizado como *bit* de paridade, e reflete a condição do registrador acumulador quanto à quantidade de números 1 existentes. Se P = 0, então a paridade é par, caso contrário a paridade é ímpar.

Os registradores de função especial P0 a P3 são utilizados para habilitar as funções especiais das portas do microcontrolador. Quando estão em nível lógico 1, suas funções especiais ficam habilitadas. A porta P0, como visto anteriormente, é utilizada para multiplexar dados e endereços, para a conexão com as memórias externas. A porta P1 é utilizada apenas como porta de entrada e saída para propósitos gerais. A porta P2 é utilizada como via de endereços mais significativos para a conexão com as memórias externas. A porta P3 pode ser configurada com funções especiais para diversos propósitos, como é mostrado na Tabela 3. Todas as portas podem ser utilizadas como entrada e saída de dados, quando suas funções especiais não estão ativas.

Os demais registradores de funções especiais não serão utilizados no programa e, por isso, não serão discutidos neste trabalho. Porém, para obter maiores informações sobre estes registradores, leia [Gimenez 2002] e [Nicolosi 2000].

Para utilizar memória externa, seja ela RAM ou ROM, é necessário o uso de um *latch* ligado ao pino de controle ALE (*Address Latch Enable*). Este *latch* ligado ao pino ALE é utilizado para demultiplexar externamente os dados e endereços. A demultiplexação é necessária, pois a porta 0 é utilizada como via multiplexada no tempo [Nicolosi 2000].

Tabela 3. Funções especiais da porta 3

Nome	Número do pino	Função especial	Significado das funções especiais
P3.0	10	RXD	Receptor de dados serial
P3.1	11	TXD	Transmissor de dados serial
P3.2	12	INT0\	Interrupção externa 0
P3.3	13	INT1\	Interrupção externa 1
P3.4	14	T0	Entrada externa do temporizador/contador 0
P3.5	15	T1	Entrada externa do temporizador/contador 1
P3.6	16	WR\	Sinalizador de escrita na memória RAM externa
P3.7	17	RD\	Sinalizador de leitura da memória RAM externa

A Figura 6 mostra como é feita a ligação da memória externa com o microcontrolador. Os pinos da porta 0 são ligados no *latch*, junto ao pino de controle ALE, e o *latch* efetua a demultiplexação, que é controlada automaticamente pelo microcontrolador, não sendo necessária qualquer intervenção do programador.

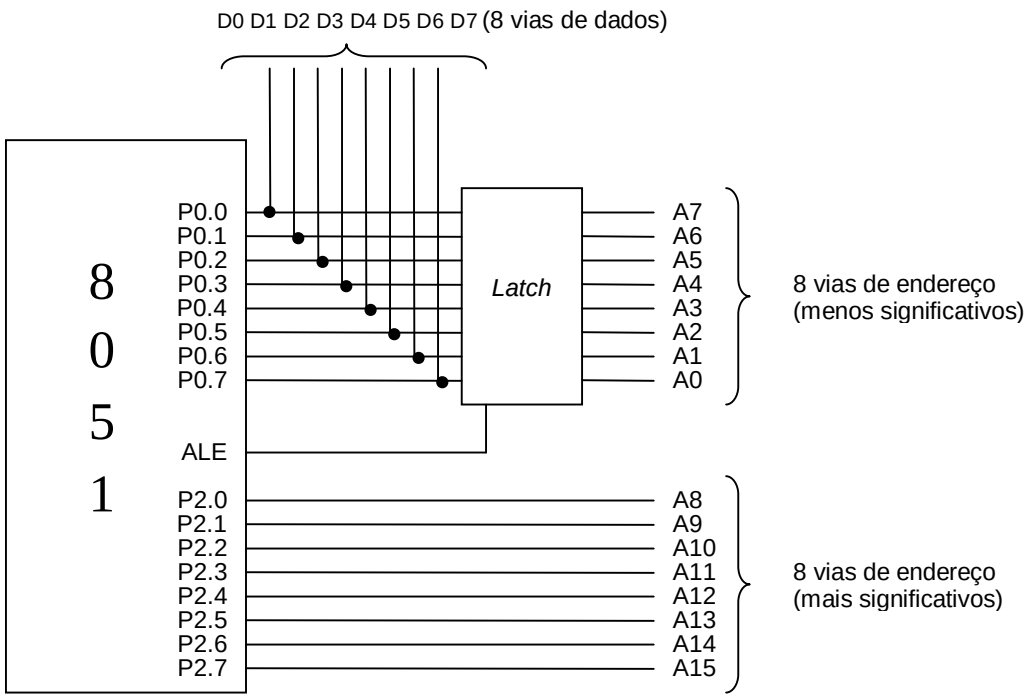


Figura 6. Demultiplexação dos dados e endereços na porta P0

Os dados utilizados pelo microcontrolador são de 8 *bits* e o endereçamento interno também. A memória externa possui um endereçamento de 16 *bits*, o que faz necessária a utilização da porta 2 como via de endereço mais significativo. Os *bits* menos significativos de endereço trafegam pela porta 0. Porém, se não for utilizada toda a capacidade de endereçamento externo, pode-se habilitar, com a função especial de acesso à memória externa, apenas a quantidade de pinos necessária para o endereçamento. Assim, deixa-se os demais pinos para sua utilização normal de entrada e saída.

A próxima seção consta da descrição do conjunto de instruções do microcontrolador 8051 utilizados no código gerado pela ferramenta, assim como os modos de endereçamento das memórias de dados e programa internas e externas.

1.2.2 Conjunto de Instruções

O microcontrolador 8051 da Intel possui um conjunto de instruções de 8 *bits* que podem executar operações de transferência de dados, aritméticas, lógicas, booleanas, de salto condicional e de salto incondicional [Gimenez 2002], [Nicolosi 2000] e [Intel 1994].

O conjunto de instruções é definido pelo fabricante do microcontrolador, e é chamada de linguagem de montagem (*Assembly*). Com esta linguagem pode-se escrever o código que será executado pelo microcontrolador. Esta linguagem é, exceto pela linguagem de máquina, de mais baixo nível possível e possui uma relação de um para um com as instruções nas linguagens de máquina [Tanenbaum 1999].

Por meio da linguagem de montagem, pode-se ter um controle total sobre os registradores e memórias de dados e programa da arquitetura-alvo e, com isso, é possível criar programas com um maior controle sobre a disposição dos dados nos registradores.

1.2.2.1 Operações de Transferência de Dados

Existem diversas maneiras de endereçar um espaço de memória. Como se viu anteriormente, as instruções responsáveis pela transferência de dados nas memórias são MOV, MOVC e MOVX. A instrução MOV é utilizada apenas para transferir dados na memória interna. A instrução MOVC é utilizada para mover dados na memória de programa e MOVX é utilizada para acessar a memória de dados externa.

Além de diferenciar as instruções para cada tipo de memória, às vezes, é necessária a utilização de modos de endereçamento diferentes, quais sejam, direto, indireto, por registrador, imediato e indexado.

No endereçamento direto, o operando é especificado por um campo de endereço de memória de 8 *bits* dentro da instrução, como por exemplo, MOV A,30h. Neste exemplo, o conteúdo do registrador acumulador receberá o conteúdo da posição de memória, cujo endereço é 30h.

No endereçamento indireto, a instrução utiliza o conteúdo do registrador R0 ou R1 do banco selecionado como um ponteiro de localização dentro de um bloco de memória (128 *bytes* internos da memória de dados). Os registradores de funções especiais não são acessados por este tipo de endereçamento. Como exemplo tem-se MOV A,@R0, onde o conteúdo do

acumulador receberá o conteúdo da posição de memória, cujo endereço é dado pelo conteúdo do registrador R0.

No endereçamento por registrador, a instrução especifica diretamente um dos registradores do banco de registradores selecionado, como por exemplo, MOV A,R2. Neste exemplo, o conteúdo do acumulador receberá o conteúdo do registrador R2. Pode-se utilizar nesta instrução todos os registradores, indicado por Rn, onde n pode variar de 0 até 7, em qualquer um dos bancos de registradores.

No endereçamento imediato, a constante referente ao dado é codificada junto à instrução, como por exemplo, MOV A,#03h. Neste exemplo, o conteúdo do acumulador receberá o valor hexadecimal 03.

No endereçamento indexado, utilizado apenas pelas instruções MOVC e JMP, o endereço de destino depende tanto do endereço inserido na instrução como do valor do acumulador. Um exemplo seria MOVC A,@A+DPTR, onde o conteúdo do acumulador receberia o conteúdo da posição de memória de programa, cujo endereço é dado pela soma do conteúdo do registrador acumulador com o conteúdo do registrador DPTR, o qual é um registrador de 16 *bits*.

1.2.2.2 Operações Lógicas

As operações lógicas são realizadas com as instruções: ANL (AND), ORL (OR), XRL (XOR), CLR (Limpa), CPL (NOT), RL (Rotaciona à esquerda), RLC (Rotaciona à esquerda por meio do *Carry Bit*), RR (Rotaciona à direita), RRC (Rotaciona à direita por meio do *Carry Bit*) e SWAP (Troca os *bits* menos significativos com os *bits* mais significativos). Estas operações são aplicadas apenas a *bytes* e não são utilizadas no programa.

1.2.2.3 Operações Booleanas

As operações booleanas são realizadas pelas instruções: ANL (AND), ORL (OR), CLR (Limpa), SETB (Atribui), CPL (NOT) e pelas instruções de salto condicionais JC, JNC, JB, JNB e JBC. Estas operações são aplicadas apenas a *bits* e apenas cinco destas são utilizadas no programa, quais sejam, CLR, SETB, CPL, JC e JNC. Estas instruções são dadas na Tabela 4.

Tabela 4. Conjunto de instruções booleanas utilizadas no programa

Instrução	*Byte*(s)	Ciclos	Descrição
CLR C	1	1	(C) ß 0
SETB bit	2	1	bit ß 1
CPL C	1	1	(C) ß not (C)
JC end	2	2	Salta para endereço da memória de programa se (C) = 1
JNC end	2	2	Salta para endereço da memória de programa se (C) = 0

1.2.2.4 Operações de Salto Incondicional

As operações de salto incondicional são utilizadas para realizar saltos na memória de programa durante a execução, sem que haja a necessidade de associar uma condição ao salto. As instruções utilizadas são: SJMP, AJMP, LJMP, JMP, ACALL, LCALL, RET, RETI e NOP, das quais se utiliza apenas as instruções AJMP e LJMP. A Tabela 5 mostra o conjunto de instruções utilizado no programa junto com os *bytes* ocupados por elas e os ciclos para execução de cada instrução.

Tabela 5. Conjunto de instruções para realizar o salto incondicional no programa

Instrução	*Byte*(s)	Ciclos	Descrição
AJMP end	2	2	Salta para um endereço de memória dentro da faixa de -1024 a 1024 *bytes* a partir do conteúdo de PC
LJMP end	3	2	Salta para um endereço de memória dentro dos 64 *Kbytes*

1.2.2.5 Operações de Salto Condicional

As operações de salto condicional são utilizadas para efetuar um salto na memória de programa de acordo com o resultado de uma condição. As instruções utilizadas são: JZ, JNZ, DJNZ e CJNE, e podem efetuar um salto dentro da faixa de -128 a 127 *bytes* a partir do endereço do PC. Estas instruções não são utilizadas no programa.

1.2.2.6 Operações Aritméticas

As operações aritméticas são realizadas por meio das instruções: ADD (Soma), ADDC (Soma utilizando o *Carry Bit*), SUBB (Subtração utilizando o *Carry Bit*), INC (Incrementa), DEC (Decrementa), MUL (Multiplicação), DIV (Divisão) e DA (Ajuste Decimal). Dentre estas instruções, utilizar-se-á apenas as instruções ADD, SUBB, INC e DEC, as quais podem ser vistas na Tabela 6 junto aos seus tempos de execução e a quantidade de *bytes* que ocupam na memória de programa.

Tabela 6. Conjunto de instruções utilizadas no programa para operações aritméticas

Instrução	*Byte*(s)	Ciclos	Descrição
ADD A, direto	2	1	(A) ß (A) + (conteúdo do endereço dado por direto)
ADD A,#data	2	1	(A) ß (A) + dado (endereçamento imediato)
SUBB A,direto	2	1	(A) ß (A) – (conteúdo do endereço dado por direto)
SUBB A,#data	2	1	(A) ß (A) – dado (endereçamento imediato)
INC direto	2	1	Conteúdo do endereço dado por direto é adicionado em 1
DEC direto	1	1	Conteúdo do endereço dado por direto é subtraído em 1

As instruções ADD e SUBB utilizam o Acumulador como um dos operandos e também para armazenar o resultado das operações. Além disso, a instrução SUBB pode ser utilizada para comparar os dois operandos utilizados, pois após sua execução, o *Carry Bit* é alterado e, se o seu valor for igual a 0, significa que o primeiro operando é maior ou igual ao segundo operando, caso contrário, se o valor do *Carry Bit* for igual a 1, significa que o primeiro operando é menor que o segundo. Esta comparação será necessária no programa-alvo e será vista no terceiro capítulo.

Após o programa ser escrito, deve-se utilizar um montador para realizar a conversão do programa na linguagem de máquina do microcontrolador. Assim que for gerado o código na linguagem de máquina, pode-se então gravar o programa na memória de programa (ROM) e executá-lo.

No decorrer desta seção foi apresentada a estrutura física e a programação do microcontrolador 8051 da Intel, o qual é a arquitetura-alvo do programa de geração de código. Na próxima seção é descrita a rede de Petri, a qual é o modelo de entrada do programa.

Também é discutida, na próxima seção, a escolha do modelo de rede de Petri utilizado no trabalho.

1.3 Redes de Petri

Redes de Petri foram desenvolvidas por Carl Adam Petri em 1962, em sua tese de doutorado, intitulada “Comunicação com autômatos”, e foi defendida na Universidade de Darmstadt, na Alemanha. A teoria geral foi desenvolvida a fim de modelar e analisar sistemas de comunicação. Porém, com o passar dos anos, sua tese foi sendo estendida em várias direções, abrangendo assim mais áreas de aplicações, tais como modelagem de protocolos de comunicação, controle de fábricas, concepção de *software* de tempo real, sistemas de informação e gerenciamento de base de dados [Cardoso e Valette 1997].

As redes de Petri podem ser definidas tanto com um formalismo matemático quanto graficamente. Na Figura 7 tem-se um modelo gráfico da rede de Petri Lugar/Transição, onde são explicitados os elementos básicos (lugar, transição, arco e marca) que constituem a rede. Os lugares podem ser interpretados como um estado da rede, informando alguma condição. As transições representam as ações da rede. Os arcos são utilizados para conectar os lugares às transições. As marcas são utilizadas para indicar se uma condição associada ao lugar é verdadeira.

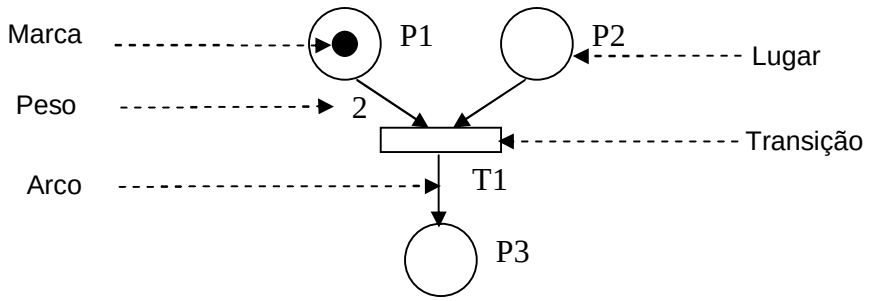


Figura 7. Exemplo de uma rede de Petri Lugar/Transição

A rede de Petri da Figura 7 pode ser descrita de forma matemática como uma quíntupla PN = (P, T, F, W, M_0), onde:

- P = {P1, P2, P3} são os lugares da rede;
- T = {T1} é a transição da rede;
- $F \subseteq (P \times T) \cup (T \times P)$ são os arcos da rede;
- W: F **à** N^+ é o peso dos arcos; e
- M_0 é a marcação inicial.

Os arcos não podem ter nas extremidades dois elementos básicos do mesmo tipo, como por exemplo, um lugar ligado a outro. Além disso, os arcos podem conter pesos, os quais informam a quantidade de marcas que devem ser retiradas dos lugares de entrada das transições ou a quantidade de marcas que devem ser criadas nos lugares de saída das transições.

Para que uma transição esteja habilitada a disparar, é necessário que os lugares de entrada (também chamados de pré-conjunto) possuam a quantidade de marcas necessárias informada nos arcos que vão do lugar para a transição. Quando a transição está habilitada, ela pode ser disparada a qualquer instante, e quando é disparada, as marcas dos lugares de entrada são removidas, de acordo com o peso dos arcos que ligam os lugares nas transições, e são criadas marcas nos lugares de saída (também chamados de pós-conjunto), de acordo com o peso dos arcos que ligam a transição aos lugares. Na Figura 7, a transição T0 estará habilitada a ser disparada quando o lugar P1 contiver pelo menos 2 marcas e o lugar P2 contiver pelo menos uma marca. Quando habilitada, ao disparar a transição, duas marcas do lugar P1 serão removidas, uma marca no lugar P2 será removida e uma marca será criada no lugar P3.

Existem hoje várias extensões da rede de Petri original, tais como as redes de Petri Temporais, Coloridas e Orientadas a Objetos. As redes de Petri Coloridas [Jensen 1997] permitem a criação de tipos de dados complexos para as marcas, o que nos permite simplificar alguns modelos de rede de Petri Lugar/Transição. Porém, a conversão de uma rede de Petri Colorida para a linguagem *Assembly* torna-se mais difícil devido à sintaxe da rede. Para uma

demonstração, compara-se o código gerado de uma mesma rede de Petri feita em dois modelos diferentes, quais sejam, Lugar/Transição e Colorida.

A rede de Petri Colorida da Figura 8 representa o problema clássico do jantar dos filósofos[1]. Neste ponto, pretende-se mostrar que a rede de Petri Colorida da Figura 8 é mais simples e fácil de compreender que a rede de Petri Lugar/Transição da Figura 9. Porém, o código gerado para a rede de Petri Colorida, mostrado no Anexo I ao final do trabalho, é significativamente maior que o código gerado para a rede de Petri Lugar/Transição, pelo fato de que os lugares nas redes de Petri Coloridas precisam de rotinas para armazenar de forma dinâmica os dados, ao contrário dos lugares nas redes de Petri Lugar/Transição, que são estáticos e são definidos durante a geração do programa.

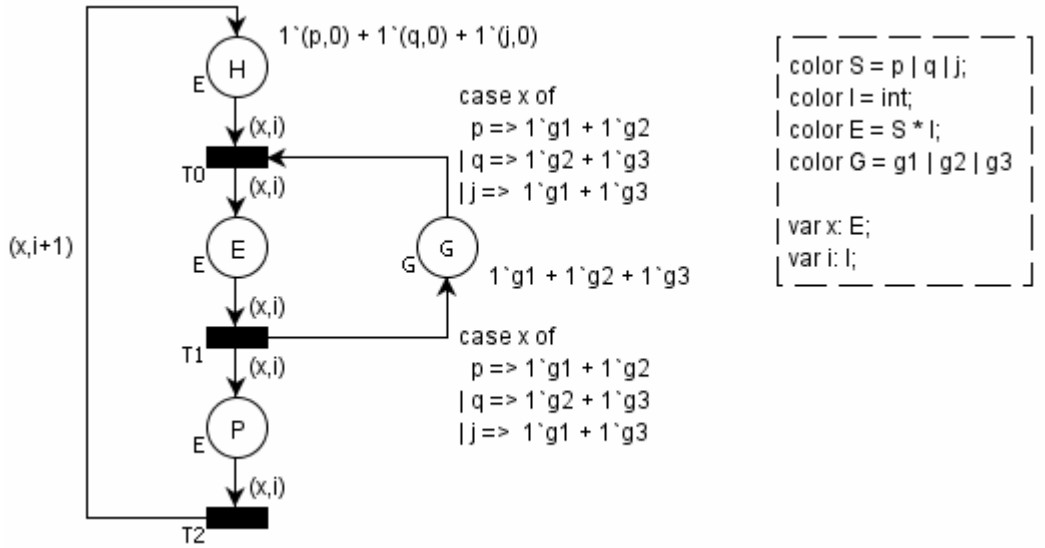


Figura 8. Exemplo de Rede de Petri Colorida do Jantar dos Filósofos

Pode-se concluir que, mesmo tendo um modelo mais compacto, a rede de Petri Colorida, quando transformada em código *Assembly* é consideravelmente maior que o código *Assembly* gerado para a rede de Petri Lugar/Transição, pelo fato de que a rede de Petri Colorida utiliza a linguagem CPNML, que permite a criação de estruturas dinâmicas na rede,

[1] Neste problema três filósofos para se alimentar necessitam dois de três talheres disponíveis em uma mesa. Eles pegam um talher de cada vez e aquele que teve sucesso obtendo os dois talheres come. Os demais aguardam o que está comendo liberar os talheres. Este problema é descrito com mais detalhes na seção 4.1.

o que aumenta consideravelmente o código pela criação de rotinas de tratamento de alocação dinâmica. O tamanho do código ocupado pelo programa gerado a partir do modelo de rede de Petri Lugar/Transição é 216 *bytes*, enquanto o tamanho do programa gerado a partir do modelo de rede de Petri Colorida é 233 *bytes*.

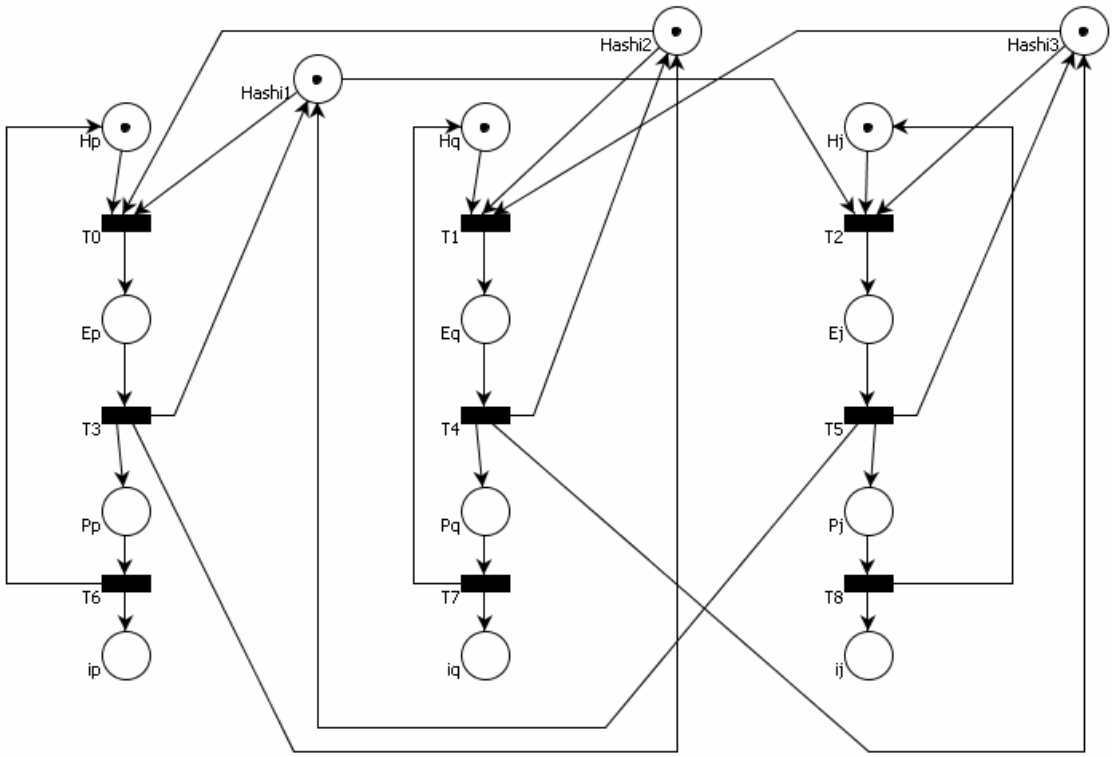


Figura 9. Rede de Petri Lugar/Transição do Jantar dos Filósofos

Para se trabalhar com a rede de Petri Lugar/Transição, utiliza-se o código PNML (*Petri Net Markup Language*), o qual é uma extensão da linguagem XML (*eXtensible Markup Language*) para descrição de redes de Petri. Neste trabalho, este código é gerado pelo programa PIPE, o qual possibilita descrever e simular redes de Petri Lugar/Transição.

A Figura 10 mostra o código PNML referente ao modelo da rede dado na Figura 7, no qual pode-se observar a declaração dos lugares entre as *tags* `<place>` e `</place>`, as declarações das transições entre as *tags* `<transition>` e `</transition>` e a declaração dos arcos entre as *tags* `<arc>` e `</arc>`.

Nas declarações das transições tem-se apenas o “id” da transição como um dado relevante para o programa, o qual pode ser encontrado dentro da *tag* `<transition>`.

```
<?xml version="1.0" encoding="iso-8859-1"?>
<pnml>
        <place id="P0">
              <name>
                    <value>P0</value>
              </name>
              <initialMarking>
                    <value>1</value>
              </initialMarking>
        </place>
        ...
        <transition id="T1">
              <name>
                    <value>T1</value>
              </name>
        </transition>
        <arc id="P0 to T1" source="P0" target="T1">
              <inscription>
                    <value>2</value>
              </inscription>
        </arc>
        ...
</pnml>
```

Figura 10. Código PNML referente a rede de Petri da Figura 7

Nas declarações dos lugares tem-se como dados relevantes para o programa o “id” gerado pela ferramenta PIPE contido na *tag* `<place>` e as marcações iniciais encontradas entre as *tags* `<initialMarking>` e `</initialMarking>`.

Nas declarações dos arcos há três dados relevantes para o programa, quais sejam: peso do arco, encontrado entre as *tags* `<inscription>` e `</inscription>`, o elemento da rede que está na sua fonte, encontrado dentro da *tag* `<arc>` e reconhecido através da palavra-chave “*source*”, e o elemento que está no destino do arco, encontrado também dentro da *tag* `<arc>` e reconhecido através da palavra-chave “*target*”.

1.4 Trabalhos Relacionados

Existem diversas ferramentas que geram código a partir de um modelo em rede de Petri, seja ela de alto ou baixo nível. Dentre estas ferramentas, as mais conhecidas são ACDC, LOOP++, CMI-PN e Artifex, as quais serão descritas abaixo e comparadas à ferramenta de geração de código desenvolvida neste trabalho e titulada ACG8051 (*Automatic Code Generation for Intel’8051*).

ACDC (*Automatic Code Generation from Design/CPN*) é um projeto que visa desenvolver técnicas e ferramentas para a geração automática de código a partir de modelos de rede de Petri Coloridas. O código gerado pela ferramenta é ML, a qual por meio de um compilador, SML/NJ [SML/NJ 2005], é convertido para código de máquina para ser executado. Este projeto constitui tanto do gerador de código como de um simulador para a rede de Petri [Mortensen 1999], ao contrário da ferramenta ACG8051, que possui apenas a geração de código e recebe um modelo já analisado de outras ferramentas, como o PIPE, que gera o código PNML. Uma vantagem que o programa em questão possui é que ele gera o código diretamente para uma arquitetura-alvo, no caso o microcontrolador 8051 da Intel, e por isso, é possível trabalhar diretamente com o seu conjunto de instruções.

LOOPN++ utiliza o modelo de rede de Petri de alto nível orientado a objeto e contém um tradutor, que gera um código na linguagem de programação C++ referente à rede de Petri. O programa gerado deve ser então compilado para ser executado [Lakos e Keen 1994]. Assim como a ACG8051, o LOOPN++ possui apenas um compilador e não possui um simulador, porém, ao contrário da ACG8051, o código gerado pelo LOOPN++ não tem relação de comportamento direta com o modelo de rede de Petri.

CPN-AMI é uma ferramenta CASE baseado em alguns tipos de redes de Petri de alto nível. Este ambiente possui um simulador e um gerador de código ADA [CPN-AMI 2005]. Assim como LOOPN++ e ao contrário da ferramenta ACG8051, CPN-AMI não possui um forte relacionamento entre a rede simulada e o código gerado. Porém, o gerador de código é capaz de gerar modelos de rede de Petri separados em processos, o que permite a execução do código em um ambiente de multi-processadores. Esta característica não é suportada pela ferramenta ACG8051.

Artifex é uma ferramenta similar a CPN-AMI, pois possui um simulador e um gerador de código, porém o código é gerado na linguagem C/C++ [Artifex 2005]. A geração de

código para uma linguagem de alto ou médio nível acarreta a necessidade de compilação, para então executar o código. A ferramenta ACG8051 gera um código na linguagem de montagem, o qual precisa ser montado para ser executado, porém, como esta linguagem é de baixo nível, sendo superior apenas à própria linguagem de máquina, pode-se ter um controle total das memórias e registradores.

Neste capítulo estudou-se o ambiente no qual o programa criado deve ser integrado, a arquitetura-alvo para a qual o código é gerado e o conjunto de instruções utilizados. Estudaram-se as redes de Petri Lugar/Transição e o porquê de sua utilização no programa ao invés da rede de Petri Colorida. Foram analisados também os trabalhos existentes relacionados com a ferramenta e as vantagens e desvantagens sobre eles. No próximo capítulo descreve-se a estrutura do programa por meio da UML (Linguagem de Modelagem Unificada).

2 CONCEPÇÃO DO PROGRAMA

Com os objetivos de reduzir o tempo gasto na etapa de síntese de *software*, explicada no primeiro capítulo, eliminar os erros da escrita manual do código e obter um programa que ocupa o menor espaço possível na memória de programa, desenvolve-se uma ferramenta para a geração automática de código para sistemas embutidos a partir de um modelo formal descrito em rede de Petri Lugar/Transição.

Neste capítulo, é descrita a estrutura e o comportamento do programa por meio da modelagem feita utilizando UML.

2.1 Modelagem do Programa

Para a realização da modelagem do programa utiliza-se UML, que é uma linguagem visual utilizada para a modelagem de sistemas computacionais, por meio do paradigma de orientação a objetos [Larman 2004].

Durante muitos anos, o termo “orientado a objetos” foi usado para denotar uma abordagem de desenvolvimento de *software* que usava uma das várias linguagens orientadas a objetos, tais como Java, C++, Smalltalk [Pressman 2002]. Porém, nos dias atuais o paradigma de orientação a objetos abrange todo o processo de engenharia de *software*.

A programação orientada a objetos tem como principais objetivos reduzir a complexidade no desenvolvimento de *software* e aumentar sua produtividade, por meio da reutilização de código e aplicação de conceitos de herança e polimorfismo. A análise, o projeto e a programação orientados a objetos são as soluções para o aumento da complexidade dos ambientes computacionais que se caracterizam por sistemas distribuídos em redes, em camadas e baseados em interfaces gráficas. Porém, para tirar um maior proveito deste paradigma, é necessário que ele seja adotado desde o início do processo de engenharia de

software até a codificação do programa. Nesta última etapa utiliza-se uma linguagem orientada a objetos tal como o Java ou C++. Uma comparação entre estas duas linguagens será feita no próximo capítulo, onde define-se qual a linguagem mais apropriada para o desenvolvimento da ferramenta.

2.1.1 Diagrama de Casos de Uso

O diagrama de casos de uso, dado na Figura 11, é o diagrama mais abstrato, informal e flexível da UML. Ele é utilizado para demonstrar o comportamento externo do sistema na visão do usuário, demonstrando as funções que o usuário poderá utilizar [Larman 2004].

Figura 11. Diagrama de Casos de Uso

O diagrama de caso de uso do sistema é simples, composto apenas por um ator, o qual se refere ao projetista do sistema, e um caso de uso, o qual se refere à função de gerar o código.

2.1.1.1 Caso de Uso GerarCódigo: Fluxo Normal

O fluxo normal do sistema é realizado caso não haja nenhum problema durante o processo de geração de código e é composto pelos seguintes passos:

1. Ator digita o arquivo de entrada, incluindo o caminho do arquivo no sistema.

2. Ator digita o arquivo de saída, incluindo o caminho do arquivo no sistema.

3. Ator digita “-i” para indicar o uso da memória de programa interna, ou digita “-e” ou deixa oculto para indicar o uso de memória de programa externa.

4. Sistema gera o código *assembly* da correspondente rede de Petri definida no arquivo de entrada e grava este código no arquivo de saída definido pelo Ator.

2.1.1.2 Caso de Uso GerarCódigo: Fluxo Alternativo

1-2a. Arquivo inválido:

1. Caso o Ator digite o nome de um arquivo inválido o sistema finaliza o processo de geração de código e retorna uma mensagem de erro.

3a. Tipo de Memória inválida:

1. Caso o Ator digite um texto diferente de “-i” ou “-e” ou deixe de passar este parâmetro, o sistema informa um erro ao Ator e finaliza o processo de geração de código.

4a. Uso de memória externa e lugares representando acesso externo:

1. Caso o Ator defina os lugares do modelo de rede de Petri como sendo sinais de controle, e também defina o uso da memória externa, o sistema informa um erro e finaliza o processo de geração de código caso as portas de acesso externo sejam as mesmas utilizadas para a conexão das memórias externas.

2.1.2 Diagrama de Classes

O diagrama de classes, dado na Figura 12, serve como base para outros diagramas da UML e define a estrutura das classes utilizadas pelo sistema, determinando os atributos e métodos possuídos de cada classe, além de estabelecer como as classes se relacionam e trocam informações entre si [Larman 2004].

O programa é composto por seis classes, quais sejam, Arco, Transicao, Lugar, CodigoFonte, Codigo e Acg8051. A classe Acg8051 é marcada com o estereótipo <<boundary>> indicando que esta classe serve de comunicação entre os atores externos e o sistema, conseqüentemente, esta classe contém o método principal para a execução do sistema. O nome Acg8051 pertence ao programa e é uma abreviação de *Automatic Code Generation for 8051’s Intel.*

Na classe Acg8051 tem-se três atributos, quais sejam, o arquivo de entrada, o arquivo de saída e a memória. Dentre eles, os dois primeiros são do tipo *File* e o último é do tipo *string*. Tem-se, também, na classe, sete métodos que são responsáveis por abrir o arquivo de entrada, criar o arquivo de saída, determinar o tipo de memória e executar o programa.

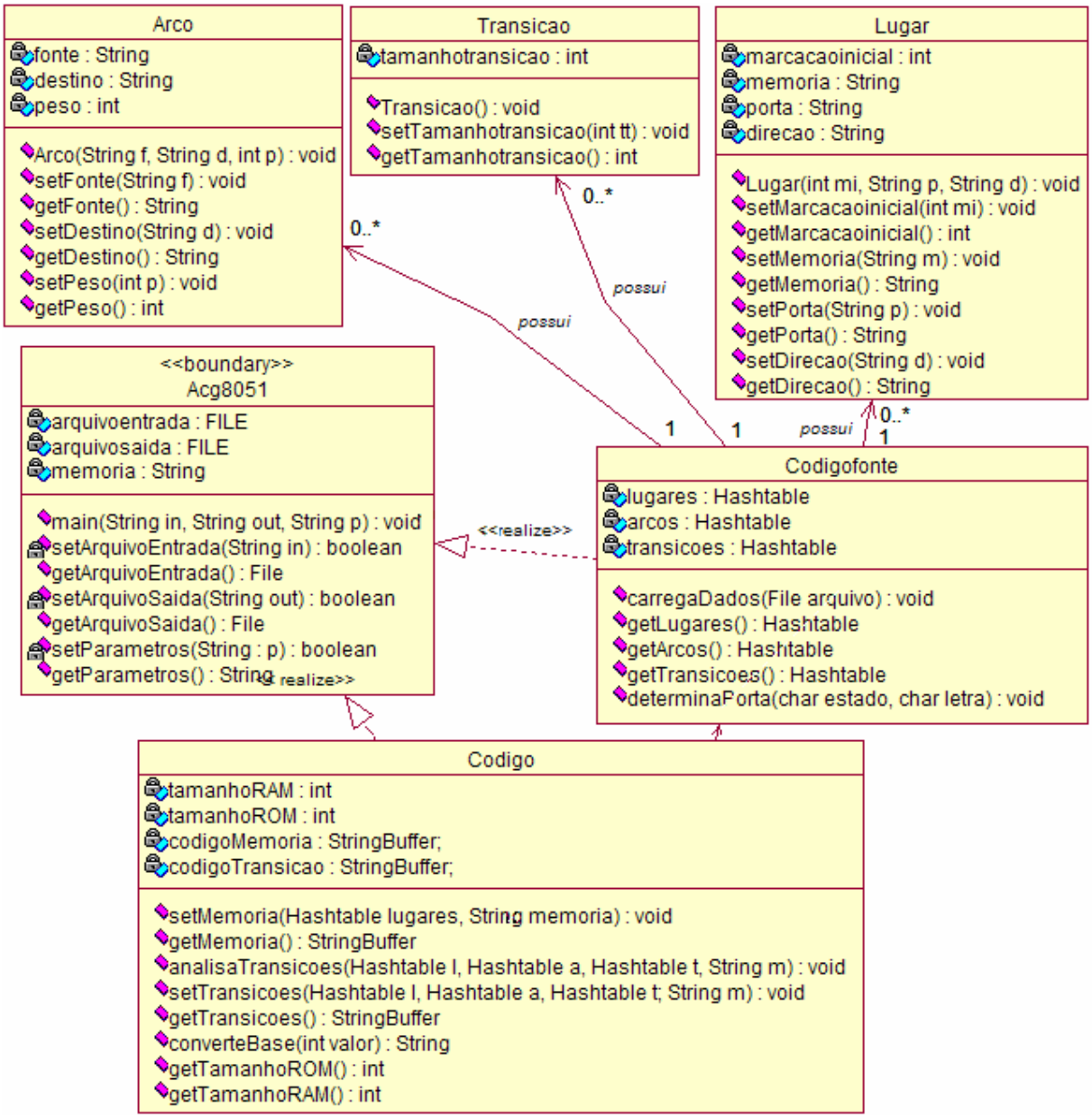


Figura 12. Diagrama de Classes

Há duas classes ligadas à classe Acg8051 por meio de um relacionamento do tipo realização. Este relacionamento indica que as classes na fonte do arco (CodigoFonte e Codigo) são responsáveis por executar funções para a classe que representa a interface do sistema, encontrada no destino do arco (Acg8051).

A classe CodigoFonte contém três atributos do tipo *Hashtable*, quais sejam, lugares, arcos e transicoes, e cinco métodos responsáveis ou por analisar o código fonte (arquivo PNML) ou retornar as *Hashtables* referentes aos lugares, arcos ou transições.

A *Hashtable* é uma tabela que permite a alocação dinâmica de elementos e é composta por uma chave e um valor, que pode ser uma classe.

Tem-se ligadas à classe CodigoFonte três classes que armazenam os dados referentes aos lugares, arcos e transições. Estas classes são gravadas como valor nas *Hashtables* respectivas, que possuem como chave o rótulo dos elementos da rede. Elas são ligadas por um relacionamento do tipo Associação Binária e indica que a classe na fonte do arco pode possuir zero ou muitas classes na outra extremidade do arco, da qual é dependente.

A classe Arco contém três atributos, seis métodos que manipulam estes atributos e um construtor, o qual recebe como parâmetro os dados a serem armazenados nos atributos por meio dos seus respectivos métodos, quais sejam, setFonte(*String*), setDestino(*String*) e setPeso(int). Os três métodos restantes, getFonte(), getDestino() e getPeso() são utilizados para recuperar o valor armazenado nos atributos **fonte**, **destino** e **peso** respectivamente.

A classe Lugar contém quatro atributos, oito métodos que manipulam estes atributos e um construtor, que recebe como parâmetros o tipo de dado inteiro, que é armazenado no atributo **marcacaoinicial** por meio do método setMarcacaoinicial(int), uma *string* informando a porta utilizada para acesso externo, e uma *string* informando a direção da porta, a qual pode ser entrada ou saída. O primeiro atributo armazena a marcação inicial do lugar na rede, ou seja, a quantidade de marcas que o lugar possui inicialmente, e pode ser recuperado com o método getMarcacaoinicial(). O segundo e terceiro parâmetro serão passados apenas quando o lugar for considerado como uma porta do microcontrolador e, caso não seja, estes parâmetros serão nulos. O atributo **memória** é responsável por armazenar a posição na memória de dados do lugar onde são adicionadas ou removidas as marcas. Para gravar a

posição de memória, utiliza-se o método setMemoria(*String*) e para recuperar a posição de memória, armazenada no atributo **memória**, utiliza-se o método getMemoria().

A classe Transicao é utilizada para armazenar o tamanho do código de programa ocupado pela transição e é composta por apenas um atributo do tipo inteiro chamado **tamanhotransicao**, por dois métodos responsáveis por armazenar um dado no atributo e recuperar o dado do atributo, e um construtor. Esta classe é utilizada pela classe CodigoFonte para gerar um código mais otimizado para as transições, como descrito no próximo capítulo.

A classe Codigo é responsável por gerar o código *Assembly* para o microcontrolador 8051 da Intel e está ligada, como visto anteriormente, à classe Acg8051. A classe Codigo também está ligada à classe CodigoFonte por meio de um relacionamento do tipo dependência, ou seja, a classe Codigo precisa da classe CodigoFonte para executar seus métodos. Na classe Codigo tem-se quatro atributos, quais sejam, **tamanhoROM**, **tamanhoRAM**, **codigoMemoria** e **codigoTransicao**, dentre os quais os dois primeiros são do tipo inteiro e os dois últimos são do tipo *StringBuffer*. O tipo de dado *StringBuffer* é utilizado para armazenar grande quantidade de cadeia da caracteres.

Na classe Codigo tem-se nove métodos que realizam a geração do código *Assembly* e recuperam-no, assim como o tamanho ocupado na memória de dados (RAM) e na memória de programa (ROM). O método setMemoria(*Hashtable*, *String*) recebe a *Hashtable* **lugares** e uma *string* indicando se o tipo de memória de dados a ser utilizada é interna ou externa. Neste método são alocados os espaços de memória de dados referentes aos lugares, os quais são gravados no atributo memória por meio do método setMemoria(*String*) da classe Lugar. Além disso, este método cria o código *Assembly* referente à criação da marcação inicial dos lugares.

O método setTransicoes(*Hashtable*, *Hashtable*, *Hashtable*, *String*) recebe as três *Hashtables* da classe CodigoFonte e uma *string* que indica se o programa será gerado para o

microcontrolador que usará memória interna ou externa. Esta transição é responsável por criar o código *Assembly* referente às transições da rede, incluindo a verificação de disparo e as ações da transição. Antes de realizar a geração do código das transições, este método invoca o método analisaTransicoes(), o qual possui os mesmos parâmetros e é responsável por armazenar o tamanho das transições no atributo **tamanhotransicao** através do método setTamanhotransicao() da classe Transicao.

Por fim, na classe Codigo, tem-se dois métodos que recuperam os valores armazenados nos atributos **tamanhoRAM** e **tamanhoROM** e um método que realiza a conversão de um dado do tipo inteiro em um valor hexadecimal, que é utilizado onde os pesos dos arcos e as posições de memória são requisitados.

Nesta seção foi descrita a estrutura das classes que formam o programa. Na próxima seção é estudado o diagrama de seqüência, o qual mostra a ordem temporal em que os métodos das classes são chamados.

2.1.3 Diagrama de Seqüência

O diagrama de seqüência, dado na Figura 13, é utilizado para mostrar a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos no sistema. Este diagrama é baseado no diagrama de casos de uso, dado na Figura 11, e apóia-se no Diagrama de Classes, dado na Figura 12, para determinar os objetos das classes envolvidas, bem como os métodos disparados entre os mesmos.

Como se pode observar no diagrama de seqüência dado na Figura 13, para executar este programa de geração de código, é necessário que o ator informe o arquivo de entrada, o arquivo de saída e, se desejar, o parâmetro referente ao tipo de memória. Caso ocorra algum erro durante o processamento dos argumentos informados, uma mensagem de acordo com o erro ocorrido é retornada para o ator.

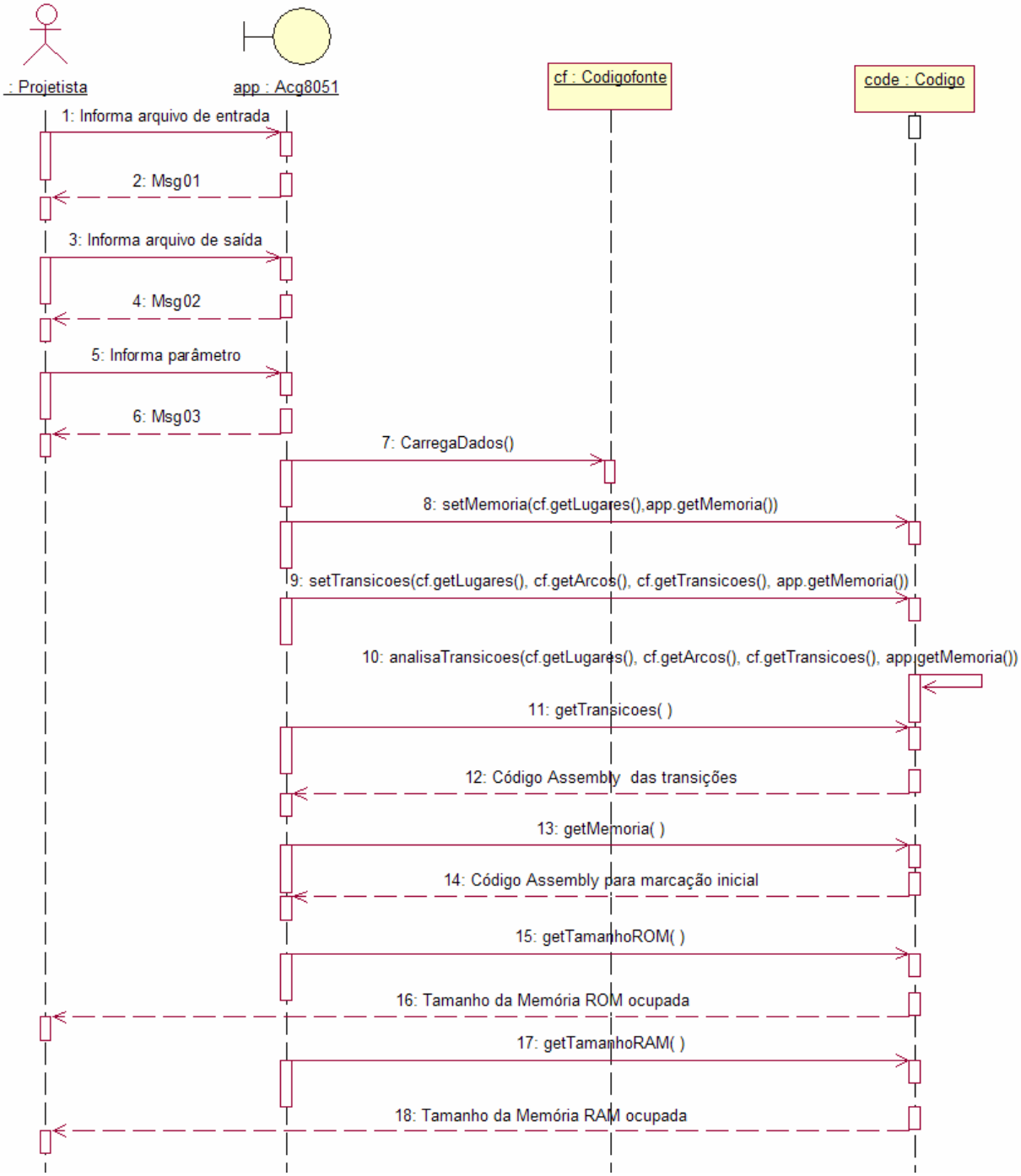


Figura 13. Diagrama de Seqüência

Há também uma instância da classe CodigoFonte, chamada **cf**, e uma instância da classe Codigo, chamada **code**. Após o ator ter informado os argumentos necessários para a execução do programa, a instância da classe Acg8051, chamada **app**, invoca o método carregaDados() do objeto **cf**, que é responsável por armazenar os dados importantes do código fonte.

Em seguida, o objeto **app** invoca o método setMemoria() do objeto **code** e informa junto, como argumentos do método, a *Hashtable* **lugares** preenchida pelo método invocado anteriormente, carregaDados (), junto à cadeia de caracteres informando o tipo de memória de dados a ser utilizada no microcontrolador, a qual foi definida pelo ator.

O próximo método invocado pelo objeto **app** é setTransicoes(), que recebe como argumentos as *Hashtables* preenchidas pelo método carregaDados () da classe CodigoFonte junto com uma cadeia de caracteres, informando o tipo de memória de dados a ser utilizada pelo microcontrolador.

Dentro do método setTransicoes(), existe uma chamada ao método analisaTransicoes(), que é responsável pelo cálculo do tamanho das transições e armazenamento destes valores na classe Transicao de cada valor da *Hashtable* **transicoes**. O método analisaTransicoes() recebe os mesmos argumentos do método setTransicoes().

Os métodos getMemoria() e getTransicoes() são chamadas pelo objeto **app** para capturar o código *Assembly* referente à criação das marcas iniciais e as verificações de disparo e ações das transições. Após capturar estes códigos, o objeto **app** salva no arquivo de saída definido pelo ator estes códigos.

Os dois últimos métodos chamados pelo objeto **app** são direcionados à classe Codigo para recuperar o tamanho utilizado pela ROM e pela RAM no código *Assembly* gerado.

Neste capítulo foi exposta a estrutura e o comportamento das classes do programa. No próximo capítulo é descrita a implementação dos métodos mencionados neste capítulo.

3 IMPLEMENTAÇÃO DO PROGRAMA

Quando se executa o sistema e, conseqüentemente, são passados os argumentos definidos no diagrama de casos de uso, tais como arquivo de entrada, arquivo de saída e parâmetro da memória, a classe Acg8051 realiza a verificação dos argumentos passados pelo ator e chama as demais classes do sistema para realizar a geração de código. Caso ocorra algum erro durante o processo de abertura ou criação de arquivos ou na análise dos parâmetros, o sistema é encerrado sem realizar a geração do código e as respectivas mensagens de erro são enviadas ao ator.

Caso os argumentos passados estejam corretos, a classe Acg8051 cria os objeto das classes que possuem os métodos para a análise do código fonte e para a geração do código, na ordem mostrada no diagrama de seqüência dado na Figura 13. Nas próximas seções são descritas as implementações das classes que analisam e geram o código.

3.1 Análise do Código Fonte

Após analisar os argumentos, a classe Acg8051 invoca o método para a análise do código, chamado carregaDados (File arquivo), o qual se encontra na classe CodigoFonte. Este método é responsável por ler o código PNML que contém a descrição da rede de Petri Lugar/Transição e armazenar os itens relevantes ao programa nas *Hashtables* lugares, transições e arcos. Tem-se nesse programa três classes que são gravadas como valor nas *Hashtables*, quais sejam, Lugares, Arcos e Transicoes.

Na *Hashtable* lugares são armazenados todos os lugares encontrados no código PNML. A chave na tabela é composta pelo rótulo do lugar, identificado no código PNML pelo trecho de código `id = “rótulo_lugar”` dentro da *tag* `<place>`. O valor na *Hashtable* é composto pela classe Lugar, a qual armazena a marcação inicial dos lugares,

identificada no código PNML pelo trecho de código `<value>marcação_inicial<value>` entre as *tags* `<initialMarking>` e `</initialMarking>`.

Na *Hashtable* arcos são armazenados todos os arcos encontrados no código PNML. A chave da tabela é composta pela cadeia de caracteres, informando a fonte concatenada com o destino do arco. O valor da *Hashtable* é composto pela classe Arco, onde é armazenada a fonte do arco, identificado pelo trecho de código `source = "fonte_arco"`, o destino do arco, identificado pelo trecho de código `target = "destino_arco"`, e o peso do arco, identificado pelo trecho de código `<value>peso_arco</value>` entre as *tags* `<inscription>` e `</inscription>`.

Na *Hashtable* transições são armazenados os rótulos das transições, os quais são, nesta etapa, os únicos dados relevantes para o programa, e são identificados no código PNML pelo trecho de código `id = "rotulo_transicao"`, dentro da *tag* `<transition>`.

Como se trabalha com código XML, a análise do código torna-se mais rápida e fácil, pois não se precisa criar um analisador para verificar cada caractere do código, mas apenas identificar as *tags* que contêm as informações necessárias para o programa e capturar as informações dentro delas.

3.2 Geração do Código *Assembly*

A classe Codigo é responsável pela geração do código *Assembly* e trabalha com as *Hashtables* preenchidas durante a execução da classe Codigofonte. Esta classe possui dois métodos responsáveis pela geração do código *Assembly*, quais sejam, setMemoria() e setTransicoes().

O primeiro método, chamado setMemoria(), é responsável por alocar os espaços na memória de dados para os lugares e criar o código *Assembly* que realizará o armazenamento das marcações iniciais nos espaços alocados anteriormente em tempo de execução. A Figura

14 mostra o algoritmo que realiza este processo de alocação de espaço e criação do código. O algoritmo lê toda a *Hashtable* lugares (linha 04) e para cada lugar, que não seja um lugar de sinal de controle, chama o método setMemoria(), o qual grava no atributo **memória** da classe Lugar o espaço de memória que o lugar assumirá no programa *Assembly* (linha 06). Em seguida, o algoritmo cria o código *Assembly* que armazena a marcação inicial dos lugares, de acordo com o tipo de memória a ser utilizada. Se a memória de dados for interna (linha 08), o código a ser gerado será referente ao uso da memória de dados interna (linha 09), caso contrário, (linha 11), o código a ser gerado será referente ao uso da memória dados externa (linhas 12, 13 e 14). De acordo com o tipo de memória, a variável **tamanhoROM**, do tipo inteiro, será atualizada com o valor do respectivo tamanho do código gerado (linha 10 e 15), ou seja, para memória interna, cada linha que realizará a alocação de marcas nos lugares ocupa 3 *bytes* e, para a memória externa, o conjunto de instruções que realizará a alocação das marcas nos lugares ocupa 6 *bytes.*

```
01    inicio
02       var contador: inteiro
03       contador = 20
04           enquanto (lugares <> vazio) faça
05              se(lugar <> sinal de controle) então
06                  lugares.setMemoria(ConverteBase(contador) + "h")
07                  contador++
08                  se (memoria = "interna") então
09                        escreva("MOV " + lugar.getMemoria() + "," +
                          ConverteBase(lugar.getMarcacaoinicial() + "h")
10                        tamanhoROM = tamanhoROM + 3;
11                  senão
12                        escreva("MOV DPTR,#" + lugar.getMemoria() + "h");
13                        escreva("MOV A,#"+ lugar.getMarcacaoinicial()+"h");
14                        escreva("MOV @DPTR,A");
15                        tamanhoROM = tamanhoROM + 6;
16                  fim-se
17              fim-se
18           fim-enquanto
19           tamanhoRAM = contador – 1;
20    fim
```

Figura 14. Algoritmo para alocação de memória de dados

O segundo método, chamado setTransicoes(), recebe como argumentos as *Hashtables* lugares, arcos e transições, além da cadeia de caracteres que informa se a memória de dados a ser utilizada no microcontrolador será interna ou externa. Este método

cria as verificações e ações de disparo referentes às transições da rede. Porém, antes da criação do código referente ao disparo das transições, o método setTransicoes() chama o método analisaTransicoes() da mesma classe, o qual grava em cada atributo **tamanhotransicao** da classe Transicao, o tamanho do código *Assembly* referente à transição, quando utilizada a instrução `JC rel`. Por meio deste valor, pode-se criar um código mais otimizado como é mostrado adiante, nesta seção.

O algoritmo da Figura 15 mostra como as transições são criadas. Este algoritmo é executado enquanto houver transições na *Hashtable* transicoes, como se pode observar na condição dada na linha 04. Dentro deste laço existe outro laço responsável por ler todos os arcos na *Hashtable* arcos (linha 06). Dentro do laço dos arcos existe uma condição responsável por verificar se a transição atual está no destino do arco (linha 07) e caso esteja, todos os lugares da *Hashtable* lugares são lidos (linha 08) e, para cada lugar dentro do laço, é feita uma verificação para determinar se o lugar corresponde à fonte do arco (linha 09) e, se verdadeiro, é criado um conjunto de instruções em *Assembly* que verificará, em tempo de execução, se a transição pode ser disparada (linha 10 a 30).

Caso a memória de dados escolhida seja a interna (linha 11), o conjunto de instruções utilizado para a verificação do disparo das transições será o de acesso à memória de dados interna (linha 12 a 14). Caso contrário, o conjunto de instruções a ser utilizado será o de acesso à memória de dados externa (linha 16 a 18).

Como dito anteriormente, o método analisaTransicoes() calcula o tamanho das transições para que se possa criar um código mais otimizado. A instrução `JC rel` realiza um salto para **rel** caso o *Carry Bit* seja zero, porém, o tamanho do salto deve ficar entre -127 e 128 *bytes* a partir do próximo *byte* do programa. Com isso, se o tamanho da transição for menor ou igual a 127 *bytes* (linha 24), utiliza-se a instrução `JC rel` (linha 25), caso

contrário, utiliza-se o conjunto de instruções `JNC rel` e `LJMP end` (linha 27 a 29), o que torna o código do programa e o tempo de execução maior.

```
01  inicio
02   var contador: inteiro;
03   contador = 0;
04   enquanto (transições <> vazio) faça
05     escreva("E" + transição + "I:")
06     enquanto (arcos <> vazio) faça
07        se (transição = destino do arco) então
08          enquanto (lugares <> vazio) faça
09              se (lugar = fonte do arco) então
10                 se(lugar <> sinal de controle) então
11                   se (memória = "interna") então
12                       escreva("MOV A," + memória do lugar)
13                       escreva("CLR C")
14                       escreva("SUBB A," + peso do arco)
15                   senão
16                       escreva("MOV DPTR,#" + memória do lugar)
17                       escreva("MOVX A,@DPTR")
18                       escreva("SUBB A," + peso do arco)
19                   fim-se
20                senão
21                    escreva("MOV C," + porta)
22                    escreva("CLP C")
23                fim-se
24                se(tamanho rotina <= 127) então
25                    escreva("JC E" + transição + "F")
26                senão
27                    escreva("JNC V" + transição + contador)
28                    escreva("V" + transição + contador + ":")
29                    escreva("LJMP E" + transição + "F")
30                fim-se
31              fim-se
32          fim-enquanto
33        fim-se
34     fim-enquanto
35     enquanto (arcos <> vazio) faça
36        se (transição = destino do arco) então
37          enquanto (lugares <> vazio) faça
38              se (lugar = fonte do arco) então
39               se(lugar = sinal de controle e direção = "saída") então
40                 escreva ("CLR " + porta)
41               senão
42                 se (memória = "interna") então
43                   se (peso do arco = 1) então
44                      escreva ("DEC" + memória do lugar)
45                   senão
46                      escreva("MOV A," + memória do lugar)
47                      escreva("SUBB A,#" + peso do arco)
48                      escreva("MOV" + memória do lugar + ",A")
49                   fim-se
50                 senão
51                     escreva("MOV DPTR,#" + memória do lugar)
52                     escreva("MOVX A,@DPTR")
53                     escreva("SUBB A,#" + peso do arco)
54                     escreva("MOVX @DPTR,A")
55                 fim-se
56               fim-se
57              fim-se
58          fim-enquanto
59        fim-se
60        se (transição = fonte do arco) então
```

Figura 15. Algoritmo para geração do código das Transições (continua)

```
61            enquanto (lugares <> vazio) faça
62                se (lugar = destino do arco) então
63                 se (lugar = sinal de controle e direção = “saída”) então
64                   escreva(“SETB ” + porta)
65                 senão
66                   se (memória = “interna”) então
67                      se (peso do arco = 1) então
68                         escreva (“INC” + memória do lugar)
69                      senão
70                         escreva(“MOV A,” + memória do lugar)
71                         escreva(“ADD A,#” + peso do arco)
72                         escreva(“MOV” + memória do lugar + “,A”)
73                      fim-se
74                   senão
75                       escreva(“MOV DPTR,#” + memória do lugar)
76                       escreva(“MOVX A,@DPTR”)
77                       escreva(“ADD A,#” + peso do arco)
78                       escreva(“MOVX @DPTR,A”)
79                   fim-se
80                 fim-se
81               fim-se
82            fim-enquanto
83          fim-se
84       fim-enquanto
85       escreva (“E” + transição + “F:” + próxima transicao)
86     fim-enquanto
87     escreva(“END”)
88   fim
```

Figura 15. Algoritmo para geração do código das Transições (continuação)

Na linha 35 está contido um laço que lê novamente todos os arcos da *Hashtable* arcos e de acordo com a posição dos lugares e transições nos arcos, o código *Assembly* para remoção de marcas nos lugares de entrada ou inserção de marcas nos lugares de saída é criado.

Caso a transição esteja no destino do arco, todos os lugares da *Hashtable* lugares são lidos (linha 36) e para cada lugar dentro do laço (linha 37) é feita uma verificação para determinar se o lugar corresponde à fonte do arco (linha 38) e, se verdadeiro, é criado um conjunto de instruções em *Assembly*, correspondente à ação da transição responsável pela remoção das marcas dos lugares de entrada (linha 39 a 55).

Se a memória de dados escolhida for a interna (linha 42), o algoritmo verifica se o peso do arco é um (linha 43) e, se verdadeiro, cria a instrução `DEC` (linha 44), caso contrário (linha 45), cria o conjunto de instruções que realiza a remoção das marcas dos lugares de entrada de acordo com o peso dos arcos (linha 46 a 48). Com isso, pode-se diminuir o

tamanho do programa, pois a instrução `DEC` ocupa apenas 2 *bytes*, enquanto o conjunto de instruções das linhas 46 a 48 ocupa 6 *bytes*.

Se a memória de dados escolhida for a externa, não há como utilizar a instrução `DEC`, portanto, simplesmente é criado o conjunto de instruções que irá realizar a remoção das marcas dos lugares de entrada da memória externa (linhas 51 a 54).

Caso a transição esteja na fonte do arco (linha 60), todos os lugares da *Hashtable* `lugares` são lidos (linha 61) e para cada lugar dentro do laço é feita uma verificação para determinar se o lugar corresponde ao destino do arco (linha 62) e, se verdadeiro, é criado um conjunto de instruções em *Assembly* correspondente à ação da transição responsável pela inserção das marcas nos lugares de saída (linha 63 a 69).

Novamente, para otimizar o programa, o algoritmo cria o código para inserção das marcas nos lugares de saída de acordo com o peso do arco. Se o peso for igual a um, utiliza-se a instrução `INC` (linha 68), caso contrário, o conjunto de instruções das linhas 70 a 72. Assim como na remoção de marcas dos lugares de entrada, quando se utiliza a memória de dados externa, não se pode utilizar a instrução `INC`, portanto, o algoritmo utilizará o conjunto de instruções dado entre as linhas 75 a 78.

Os métodos a serem disparados devem possuir uma seqüência, mostrada na Figura 13 através do diagrama de seqüência, para que os dados requisitados por um método sejam criados anteriormente por outro método.

A linguagem de programação escolhida para a implementação do programa de geração automática de código foi Java, pelas razões discutidas na próxima seção.

3.3 Definição da Linguagem de Programação

Para que o sistema possa rodar em ambiente Windows e Linux e em plataformas PC ou Macintosh, a linguagem Java é a mais apropriada, pois ela pode ser executada em qualquer plataforma ou sistema operacional que se tenha instalado a máquina virtual Java.

Porém, preocupa-se, além da portabilidade, com a eficiência do programa final e por isso, foram desenvolvidos dois programas que geram o código *Assembly* a partir da rede de Petri Lugar/Transição utilizando apenas a memória interna do microcontrolador 8051 da Intel e, como foi realizada a modelagem do programa em UML, a qual segue o paradigma de orientação a objetos, os dois programas desenvolvidos seguem este paradigma, porém, um programa é interpretado, linguagem Java, e o outro programa é compilado, linguagem C++.

Por meio dos testes feitos utilizando as redes de Petri, discutidas no próximo capítulo, criou-se a Tabela 7 que compara o tempo de execução por meio do comando "*time*" do sistema operacional Linux.

Este comando retorna o tempo de execução do programa em três modos, todos com a unidade de tempo em milisegundos. O primeiro modo refere-se ao tempo gasto desde o momento da chamada do programa até a sua completa execução. O segundo modo refere-se ao tempo gasto na execução do programa para o usuário atual e o último modo refere-se ao tempo gasto na execução do programa para o sistema.

Na primeira coluna da tabela tem-se o número da rede de Petri utilizada, as quais são detalhadas no próximo capítulo. A segunda coluna refere-se ao tempo real gasto, a terceira refere-se ao tempo do usuário e a última coluna refere-se ao tempo do sistema. Para cada programa são dadas estas três medidas.

Tabela 7. Comparação da eficiência das linguagens de C++ e Java

	Linguagem Java			Linguagem C++		
Rede de Petri	Real	Usuário	Sistema	Real	Usuário	Sistema
01	0m0.200s	0m0.015s	0m0.015s	0m0.034s	0m0.015s	0m0.015s
02	0m0.198s	0m0.015s	0m0.015s	0m0.024s	0m0.015s	0m0.015s
03	0m0.198s	0m0.015s	0m0.015s	0m0.028s	0m0.015s	0m0.015s

Pode-se observar que o tempo gasto pela UCP por usuário e pelo sistema é exatamente igual, porém os tempos reais gastos na execução do programa C++ são menores que o do programa Java, pois este precisa primeiro carregar a máquina virtual Java para então executar

o sistema. Porém, quando carregado o programa, a linguagem Java se torna relativamente tão eficiente quanto a linguagem C++.

Portanto, pode-se concluir que o uso da linguagem Java se torna a mais apropriada para a execução do sistema, uma vez que ela foi desenvolvida para ser portável para qualquer sistema operacional e *hardware* e que seu tempo de execução é relativamente semelhante aos dos programas compilados.

4 DISCUSSÃO DOS TESTES

Neste capítulo são descritos os testes realizados na ferramenta de geração automática de código. Para se realizar estes testes, foram utilizados quatro modelos diferentes de redes de Petri Lugar/Transição, os quais são discutidos neste capítulo. O programa, chamado Acg8051, foi desenvolvido utilizando a linguagem de programação Java, de acordo com o estudo realizado pela comparação das linguagens compiladas e interpretadas.

Neste capítulo é utilizado o programa final, que gera o código tanto utilizando a memória interna como a memória externa de dados do microcontrolador 8051 da Intel. É mostrado também um exemplo do uso de lugares como sendo sinais de controle, permitindo assim o acesso externo ao microcontrolador. Para realizar os testes, foi utilizado um microcomputador com processador Intel de 3.2 GHz e 512 MB de memória RAM.

4.1 Estudo de Caso 1: Jantar dos Filósofos

A rede de Petri Lugar/Transição dada na Figura 16 foi a primeira rede utilizada para a realização de testes na ferramenta de geração de código. Esta rede representa um modelo do problema dos filósofos [Tanenbaum 2001], muito utilizado na área da computação concorrente e que tem como objetivo demonstrar a alocação de recursos quando se tem processos que concorrem pela alocação dos mesmos.

O problema dos filósofos, também chamado de jantar dos filósofos, consiste de três filósofos que podem estar em três estados diferentes: comendo, pensando ou com fome. Os filósofos estão à volta de uma mesa, sendo que cada um deles tem a sua frente um hashi e um prato de comida. São, no entanto, necessários dois hashis para que um filósofo possa comer, ou seja, um filósofo precisa do seu hashi e do de seu vizinho. Um impasse ocorre quando todos os filósofos pegam um hashi e aguardam a liberação do outro hashi.

Na Figura 16, os lugares Hashi1, Hashi2 e Hashi3 representam os hashis, ou seja, os recursos do sistema, e estão inicialmente todos disponíveis. Os lugares Hp, Hq e Hj representam o estado “com fome” de cada filósofo p, q e j, e ambos estão inicialmente com uma marca. Os lugares Ep, Eq e Ej representam que os filósofos que possuem a marca estão comendo. Os lugares Pp, Pq e Pj representam que os filósofos estão pensando. Os lugares ip, iq e ij representam a quantidade de iterações que já ocorreram para cada filósofo.

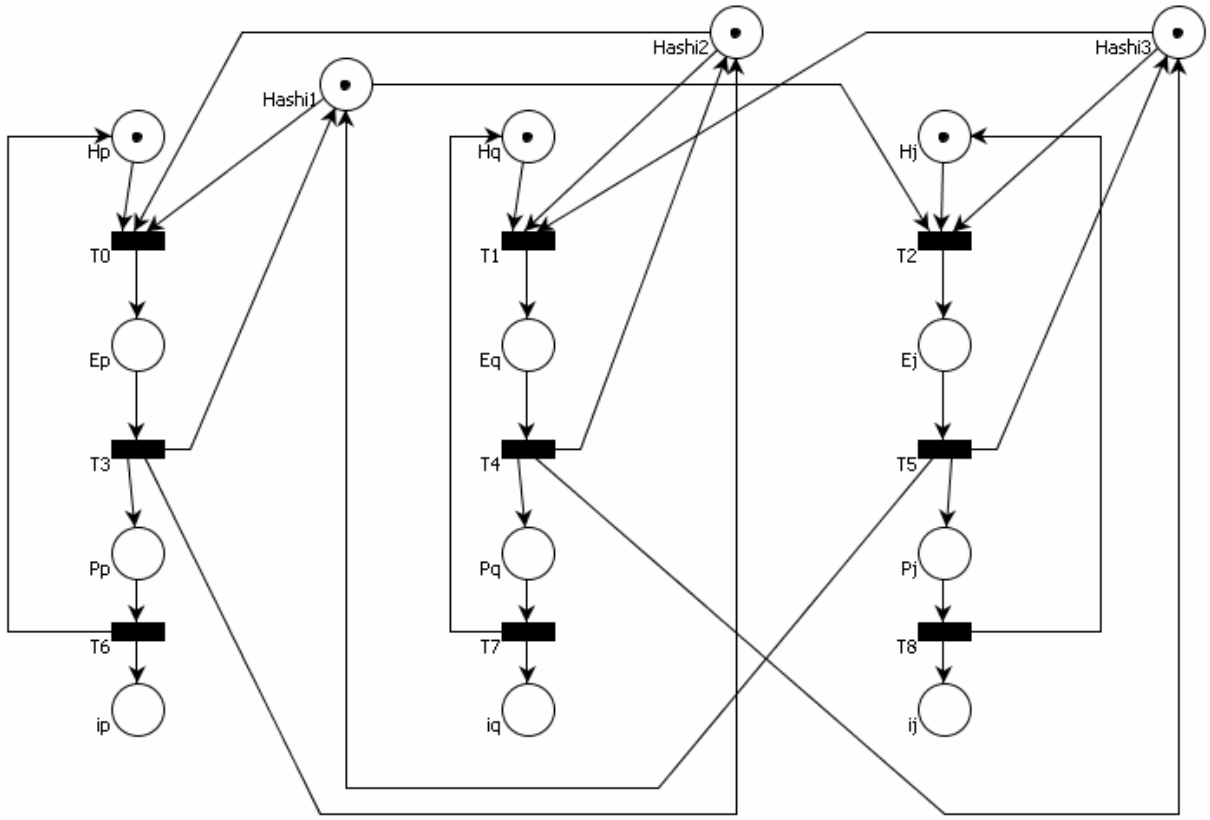


Figura 16. Modelo de Rede de Petri do problema dos filósofos

Para que o filósofo p mude seu estado de “com fome” para “comendo” a transição T0 precisa ser disparada e para isso é necessário que se tenha uma marca no lugar Hp, uma marca no lugar Hashi1 e uma marca no lugar Hashi2.

Quando disparada a transição T0, as marcas nos lugares de entrada são retiradas e uma marca no lugar Ep é criada, representando que o filósofo p está comendo. Neste ponto, os outros filósofos não poderão comer, pois não haverá hashis para eles.

Para que a transição T3 dispare, ou seja, o filósofo p mude da situação “comendo” para a situação “pensando”, é necessário que haja pelo menos uma marca no lugar Ep, ou seja, o filósofo precisa estar comendo. Quando disparada, a transição T3 remove a marca do lugar Ep, cria uma marca no lugar Pp e devolve as marcas para os lugares Hashi1 e Hashi2, ou seja, retorna os hashis para a mesa.

O mesmo acontece para os filósofos q e j. Portanto, apenas um dos três filósofos estará comendo, enquanto os outros poderão estar com fome ou pensando.

Para que a transição T6 seja disparada, é necessário que haja uma marca no lugar Pp, ou seja, a situação do filósofo p é “pensando”. Quando esta transição é disparada, a marca do lugar Pp é removida e uma nova marca é criada no lugar Hp, fazendo com que o filósofo entre na situação “com fome” novamente. Além disso, uma nova marca é criada no lugar ip, o qual é utilizado para contar a quantidade de iterações ocorridas do filósofo p.

Uma parte do código *Assembly* gerado pela ferramenta para este exemplo é dado na Figura 17. Neste trecho de código tem-se os lugares, transições e arcos referentes ao filósofo p.

O código entre as linhas 01 e 06 é referente à criação de marcas iniciais nos lugares Hp, Hq, Hj, Hashi1, Hashi2 e Hashi3. O código entre as linhas 07 e 23 é referente à verificação de disparo e à ação da transição T0, responsável pela mudança da situação “com fome” para “comendo” do filósofo p. Na linha 07 tem-se o rótulo da sub-rotina e a instrução que grava no acumulador o conteúdo da posição de memória 15h, a qual se refere o lugar Hp. Na linha 08 a instrução `CLR C` zera o *bit* C, para não afetar na subtração feita na linha 09. Esta operação subtrai 1 do acumulador, onde 1 é referente ao peso do arco que liga o lugar Hp à transição T0. Na linha 10 há uma instrução de salto condicional, o qual salta para o fim da sub-rotina da transição caso C possua o valor 0, ou seja, caso o valor do acumulador seja

menor que o peso do arco. Neste caso, a transição não pode ser disparada, pois não há marcas suficientes nos lugar de entrada Hp.

```
01     MOV 14h,#1h
02     MOV 15h,#1h
03     MOV 1Fh,#1h
04     MOV 20h,#1h
05     MOV 21h,#1h
06     MOV 22h,#1h

07     ET0I:  MOV A,15h
08            CLR C
09            SUBB A,#1h
10            JC ET0F
11            MOV A,20h
12            CLR C
13            SUBB A,#1h
14            JC ET0F
15            MOV A,21h
16            CLR C
17            SUBB A,#1h
18            JC ET0F
19            DEC 15h
20            DEC 20h
21            DEC 21h
22            INC 1Eh
23     ET0F:  LJMP ET8I

24     ET3I:  MOV A,1Eh
25            CLR C
26            SUBB A,#1h
27            JC ET3F
28            DEC 1Eh
29            INC 1Bh
30            INC 20h
31            INC 21h
32     ET3F:  LJMP ET2I

33     ET6I:  MOV A,1Bh
34            CLR C
35            SUBB A,#1h
36            JC ET6F
37            DEC 1Bh
38            INC 15h
39            INC 18h
40     ET6F:  LJMP ET5I
```

Figura 17. Trecho de código *Assembly* da rede de Petri Lugar/Transição do jantar dos filósofos

O mesmo acontece entre as linhas 11 a 14 e 15 a 18, onde o primeiro código realiza uma verificação na quantidade de marcas do lugar Hashi1 e o segundo verifica a quantidade de marcas no lugar Hashi2. Se não existirem marcas em Hashi1 ou em Hashi2, é realizado um salto para o fim da transição (ET0F), simbolizando que o Hashi1 ou o Hashi2 não está disponível para o filósofo.

Na linha 23 está a última instrução da sub-rotina para T0, a qual possui uma instrução de salto incondicional para a transição T8. A transição para a qual será feito o salto é escolhida de forma aleatória na geração do código.

O trecho de código entre as linhas 24 e 32 refere-se à transição T3. Esta transição muda a situação do filósofo p de “comendo” para “pensando”. Para que esta transição seja disparada é necessário que haja uma marca no lugar Ep, o que é verificado no código entre as linhas 24 a 27. Se não houver marcas em Ep, ou seja, o filósofo não está na situação “comendo”, ocorre um salto condicional para o fim da transição, dado na linha 27. Caso contrário, se o filósofo está na situação “comendo”, a transição T3 é disparada e a marca do lugar Ep é retirada, as marcas nos lugares Hashi1 e Hashi2 são novamente criadas (linhas 30 e 31), simbolizando que os hashis estão de volta à mesa, e uma marca no lugar Pp é criada (linha 29), simbolizando que o filósofo p está agora na situação “pensando”.

O trecho de código entre as linhas 33 e 40 é referente à transição T6. Esta transição muda a situação do filósofo p de “pensando” para “com fome”. Para que ela seja disparada é necessário que haja uma marca no lugar Pp. A verificação da existência desta marca é feita entre as linhas 33 e 36. Se não há marcas em Pp, ou seja, o filósofo p não está na situação “pensando”, um salto é feito para o fim da transição (linha 36). Caso contrário, se o filósofo está “pensando”, a transição T6 é disparada e a marca em Pp é removida, uma marca em Hp é criada e uma marca em ip também é criada, a qual simboliza a quantidade de iterações que ocorreram para o filósofo p.

4.2 Estudo de Caso 2: Sistema de Produção e Consumo

O segundo exemplo utilizado para testar o programa é dado na Figura 18. Esta rede simboliza o processo de produção e consumo. Este sistema é utilizado, por exemplo, na produção de peças automotivas, sendo que neste exemplo o consumo é feito na própria fábrica por meio da retirada da peça produzida e sua colocação no carro.

Neste sistema, a parte responsável pela produção é composta pelos lugares P0, P1 e P2 e pelas transições T0, T3 e T4. A parte responsável pelo consumo é composta pelos lugares P3, P4 e P5 e pelas transições T1, T2 e T5. O lugar P6 é considerado um depósito, onde são armazenadas as peças produzidas e de onde são retiradas as peças que serão consumidas. O lugar P7 é considerado um registro para armazenar a quantidade máxima de peças que podem ser armazenadas em estoque, no caso, três peças.

A marcação inicial da rede é dada como uma marca em P0 e uma marca em P3. Sempre que P0 possuir uma marca, simboliza que a produção da peça foi feita, e conseqüentemente, a peça pode ser armazenada no estoque. Quando se tem uma marca em P3, a parte de consumo pode ser realizada, contanto que se tenha pelo menos uma marca em P6, ou seja, para que haja o consumo da peça é necessário que esta tenha sido produzida.

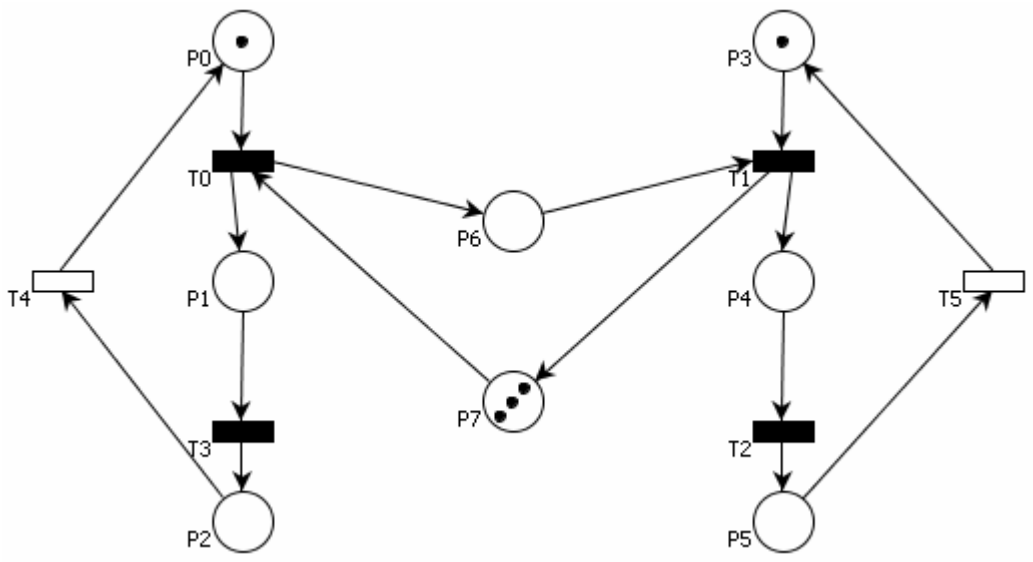


Figura 18. Rede de Petri Lugar/Transição de um modelo de produção e consumo

Para que a transição T0 seja disparada e armazene a peça produzida no estoque é necessário que haja uma marca em P0 e pelo menos uma marca em P7. Quando não houver marcas em P7, significa que o sistema já está com o estoque cheio. Quando a transição T0 é disparada, a marca de P0 é removida, uma marca de P7 é removida e uma nova marca é criada em P6, simbolizando o armazenamento no estoque da nova peça criada, e uma nova marca

também é criada em P1, simbolizando que o sistema está novamente no processo inicial de produção.

O disparo da transição T3 é feito apenas quando houver uma marca em P1. Quando disparada, a transição T3 consome a marca de P1 e cria uma nova marca em P2. O lugar P2 simboliza, de forma hipotética, que o processo de produção (transição T3) capturou a matéria prima para a construção da peça automotiva.

A transição T4 pode ser disparada apenas quando houver uma marca em P2. Quando disparada, a transição T4 remove uma marca do lugar P2 e cria uma nova marca no lugar P0. Esta transição realiza a produção da peça e a armazena em P0.

Sendo assim, se o lugar P7 estiver vazio, a peça criada não poderá ser armazenada no estoque e, portanto, o processo de produção fica em estado de espera. Quando uma peça for consumida do estoque, o processo de produção retorna ao estado ativo.

A parte de consumo é realizada a partir da transição T1. Para que esta transição possa ser disparada, é necessário que haja uma marca em P3 e pelo menos uma marca em P6, simbolizando que existe pelo menos uma peça produzida que possa ser consumida. Quando a transição T1 é disparada, a marca em P3 é consumida, uma marca em P6 é retirada, simbolizando que uma peça foi consumida, sendo uma marca criada em P7 e outra em P4. A marca criada em P7 informa que uma nova peça pode ser armazenada no estoque.

A marca em P4 simboliza que uma peça foi removida do estoque e colocada no estoque do processo de instalação da peça no carro. A partir do disparo da transição T1, o processo de instalação da peça é iniciado e só será possível retirar uma nova peça do estoque quando todo o processo tiver acabado.

Para que a transição T2 seja disparada, é necessário que haja uma marca em P4. Quando esta transição é disparada, a marca do lugar P4 é removida e uma nova marca é criada

em P5. O disparo desta transição simboliza que o equipamento de instalação pegou a peça que estava no estoque.

Para que a transição T5 seja disparada, é necessário que haja uma marca em P5. Quando esta transição é disparada, a marca do lugar P5 é removida e uma nova marca em P3 é criada. O disparo da transição simboliza a instalação da peça que está contida no equipamento, no carro.

Após o disparo da transição T5, a parte de consumo retorna ao estado inicial e fica aguardando que haja uma peça em estoque, lugar P6, para que reinicie todo o processo.

Na Figura 19 é mostrado o trecho de código referente às transições T0 e T1. Escolheram-se estas duas transições por serem a parte fundamental no processo de produção e consumo. Pode-se perceber que primeiro há a transição T1 e depois a transição T0. Isto se deve ao fato de que as transições, quando habilitadas, podem disparar em qualquer ordem.

O código entre as linhas 01 e 03 é referente à criação das marcações iniciais nos lugares P0 (15h), P3 (1Ah) e P7(16h). Os lugares P0 e P3 recebem uma marca e no lugar P7 são criadas três marcas.

O trecho de código entre as linhas 04 e 16 é referente à verificação e ao disparo da transição T1. Entre as linhas 04 e 06 é verificado se há uma marca em P6 (17h), por meio da instrução `SUBB`. Quando se executa esta transição, o valor do *Carry Bit* é alterado para zero se o primeiro operando (A) é maior ou igual ao segundo operando (1h) e, caso contrário, é alterado para um. Com isso, pode-se determinar se há a quantidade de marcas necessária nos lugares de entrada, dado pelo peso do arco. Na linha 07 há uma instrução `JC`, a qual realiza um salto para o endereço de memória de programa, dado pelo rótulo “ET1F”, caso o valor do *Carry Bit* seja um, isto é, se a quantidade de marcas no lugar de entrada é menor que a quantidade de marcas necessária para o disparo da transição informada no peso do arco.

Caso existam marcas suficientes para o disparo da transição e, conseqüentemente, o valor de *Carry Bit* seja zero, o restante do trecho de código referente à transição T1 é executado, caso contrário, é realizado um salto para a última instrução da transição (linha 16), a qual salta incondicionalmente para a posição de memória dada pelo rótulo “ET0I”.

```
01    MOV 15h,#1h
02    MOV 16h,#3h
03    MOV 1Ah,#1h

04    ET1I:  MOV A,17h
05           CLR C
06           SUBB A,#1h
07           JC ET1F
08           MOV A,1Ah
09           CLR C
10           SUBB A,#1h
11           JC ET1F
12           DEC 17h
13           DEC 1Ah
14           INC 16h
15           INC 19h
16    ET1F:  LJMP ET0I

17    ET0I:  MOV A,15h
18           CLR C
19           SUBB A,#1h
20           JC ET0F
21           MOV A,16h
22           CLR C
23           SUBB A,#1h
24           JC ET0F
25           DEC 15h
26           DEC 16h
27           INC 17h
28           INC 14h
29    ET0F:  LJMP ET5I
```

Figura 19. Trecho de código *Assembly* da rede de Petri Lugar/Transição da produção e consumo

O trecho de código dado entre as linhas 08 e 11 realiza o mesmo processo das linhas 04 a 07, exceto pelo fato de que o lugar de entrada agora é P1 (1Ah).

O código das linhas 12 e 13 realiza a remoção das marcas dos lugares de entrada, P6 e P3, respectivamente. O código da linha 14 realiza a inserção de uma marca no lugar P7 (16h) e o código da linha 15 realiza a inserção de uma marca no lugar P4 (19h).

Se ambas as condições de disparo das transições forem satisfeitas (linha 04 a 11), a transição é disparada (linha 12 a 15). Após ser executada, a linha 16 realiza um salto para a

transição T0, a qual realiza o mesmo processo de verificação e disparo, porém agora da transição T0.

O trecho de código entre as linhas 17 e 20 realiza a verificação de marcas no lugar de entrada P0 e, caso não haja a quantidade de marcas suficientes, é realizado um salto para "ET0F" na linha 20. O trecho de código entre as linhas 21 e 24 realiza a mesma verificação de marcas, porém no lugar de entrada P7.

As linhas 25 e 26 realizam a remoção de uma marca nos lugares P0 e P7, respectivamente. As linhas 27 e 28 realizam a inserção de uma marca nos lugares P6 e P1, respectivamente. Na linha 29 tem-se uma instrução de salto incondicional (`LJMP`) que realiza um salto para o endereço de memória de programa dado pelo rótulo "ET5I".

4.3 Estudo de Caso 3: Sistema de Escrita e Leitura

A rede de Petri utilizada neste exemplo é dada na Figura 20. Neste modelo, supõe-se um sistema aplicado à escrita e à leitura de arquivos em computador. Neste sistema é possível ler até três arquivos ao mesmo tempo, porém, para escrever em um arquivo, não se pode ler, simultaneamente, nenhum outro arquivo [Tanenbaum 2001].

Os lugares P0 e P1 e as transições T0 e T1 correspondem à parte do sistema que lê os arquivos. Os lugares P3 e P4 e as transições T2 e T3 correspondem à parte do sistema que escreve em um arquivo. O lugar P2 é utilizado para controlar a escrita e a leitura, sendo que, para escrever em um arquivo é necessário que haja três marcas no lugar P2 e para ler um arquivo é necessário que haja pelo menos uma marca em P2.

A transição T0 é responsável por capturar o arquivo no sistema e armazená-lo no *buffer* de leitura (P1). Esta transição será disparada apenas quando houver pelo menos uma marca nos lugares P2 e P0. Quando disparada, esta transição remove uma marca do lugar P0 e insere uma nova marca no lugar P1. Enquanto a marca estiver em P1, o usuário do sistema poderá ler o arquivo.

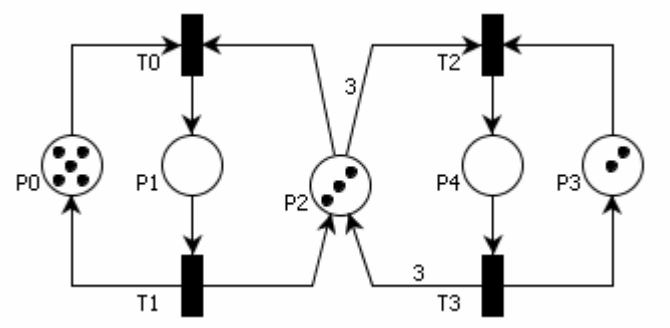


Figura 20. Modelo de Rede de Petri Lugar/Transição para um sistema de escrita e leitura

A transição T1 é responsável por capturar o arquivo do *buffer* e liberá-lo no sistema novamente. Para que esta transição possa ser disparada, é necessário que haja pelo menos uma marca em P1. Quando esta for disparada, uma marca do lugar P1 será removida e uma marca será criada nos lugares P0 e P2.

A transição T2 será disparada apenas quando nenhum arquivo estiver sendo lido, ou seja, o lugar P2 deverá conter três marcas. Quando esta transição é disparada, as três marcas do lugar P2 são removidas, impedindo que o sistema possa ler qualquer arquivo, uma marca do lugar P3 também é removida e uma nova marca é criada no lugar P4. Enquanto houver uma marca no lugar P4, o usuário do sistema poderá salvar o arquivo.

A transição T3 é responsável por liberar o arquivo que está em estado de escrita. Para que esta transição seja disparada é necessário que haja uma marca no lugar P4. Quando for disparada, esta transição cria uma marca em P3 e três marcas em P2, liberando assim o sistema para a leitura de arquivos novamente.

Na Figura 21 é mostrado o trecho de código referente à transição T2 da rede da Figura 20. Para efeito de demonstração completa do sistema, o código apresentado foi gerado pela ferramenta e foi definido o uso da memória de dados externa. O programa poderia ter sido gerado para memória de dados interna, de forma semelhante.

O trecho de código entre as linhas 01 e 09 é referente à criação das marcações iniciais nos lugares P0 (15h), P3 (17h) e P2 (18h), respectivamente. Como se utiliza a memória de

dados externa, o conjunto de instruções para acesso a este tipo de memória muda. O endereço de memória de dados externa é dado pelo registrador de 16 *bits* chamado DPTR (*data pointer*). A linha 01 armazena o valor 15, em hexadecimal, no registrador DPTR. A linha 02 armazena no acumulador o valor 5, em hexadecimal. A linha 03 armazena no conteúdo da posição de memória dada pelo registrador DPTR o valor do acumulador. É importante observar que a instrução utilizada para acessar a memória de dados externa é MOVX, e não mais MOV, a qual realiza a transferência de dados apenas na memória interna.

O processo de verificação da quantidade de marcas nos lugares de entrada para o disparo da transição é o mesmo. O que diferencia é apenas o modo como se acessa a memória de dados.

```
01    MOV DPTR,#15h P0
02    MOV A,#5h
03    MOVX @DPTR,A
04    MOV DPTR,#17h P3
05    MOV A,#2h
06    MOVX @DPTR,A
07    MOV DPTR,#18h P2
08    MOV A,#3h
09    MOVX @DPTR,A

10    ET2I:  MOV DPTR,#18h
11           MOVX A,@DPTR
12           CLR C
13           SUBB A,#3h
14           JC ET2F
15           MOV DPTR,#17h
16           MOVX A,@DPTR
17           CLR C
18           SUBB A,#1h
19           JC ET2F
20           MOV DPTR,#18h
21           MOVX A,@DPTR
22           SUBB A,#3h
23           MOVX @DPTR,A
24           MOV DPTR,#17h
25           MOVX A,@DPTR
26           SUBB A,#1h
27           MOVX @DPTR,A
28           MOV DPTR,#16h
29           MOVX A,@DPTR
30           ADD A,#1h
31           MOVX @DPTR,A
32    ET2F:  LJMP ET1I
```

Figura 21. Trecho de código *Assembly* da rede de Petri Lugar/Transição de leitura e escrita

O trecho de código entre as linhas 10 e 32 é referente à transição T2. Neste trecho de código há duas verificações, uma entre as linhas 10 e 14, a qual verifica se há marcas necessárias no lugar P2, e outra entre as linhas 15 e 19, a qual verifica se há marcas necessárias no lugar P3. Caso haja a quantidade de marcas necessárias para o disparo da transição nos lugares de entrada, a transição executa o trecho de código entre as linhas 19 e 32, caso contrário ocorre um salto para a linha 32 que contém um salto incondicional para a transição T1.

O trecho de código entre as linhas 20 e 23 realiza a remoção das três marcas do lugar P2. A instrução `MOV DPTR,#18h` armazena em DPTR o valor hexadecimal dezoito, referente à posição de memória do lugar P2. A instrução `MOVX A,@DPTR` armazena no acumulador o valor da posição de memória dada por DPTR, no caso, 18h. A instrução `SUBB A,#3h` subtrai do acumulador o valor hexadecimal três. E por fim, a instrução `MOVX @DPTR,A` armazena o novo valor no endereço de memória dado por DPTR.

O trecho de código entre as linhas 24 e 27 realiza o mesmo processo descrito no parágrafo anterior, exceto pelo fato de que apenas uma marca é removida e o lugar de entrada é P3.

Por fim, a transição realiza a inserção de uma marca no lugar P4, que é feita pelas instruções dadas entre as linhas 28 e 31. O processo é o mesmo descrito nos parágrafos anteriores, exceto pelo fato de que a instrução utilizada não é mais `SUBB` e sim `ADD` (linha 30).

4.4 Estudo de Caso 4: Sistema de Controle de Temperatura

Neste estudo de caso é apresentado um sistema simples de controle de temperatura, apenas para ilustrar, de forma didática, a potencialidade do uso do programa para sistemas de controle e automação utilizando comunicação externa. Este sistema liga ou desliga um determinado equipamento para condicionamento do ar, como, por exemplo, um ventilador, de

acordo com a faixa de valor da temperatura medida através de um sensor de temperatura TMP.

Além do microcontrolador 8051 da Intel e do sensor de temperatura, é necessária a utilização de um conversor Analógico/Digital (A/D), um ventilador e um circuito inversor, comercialmente conhecido como TTL 7404. Na Figura 22 é apresentado o diagrama em blocos do circuito.

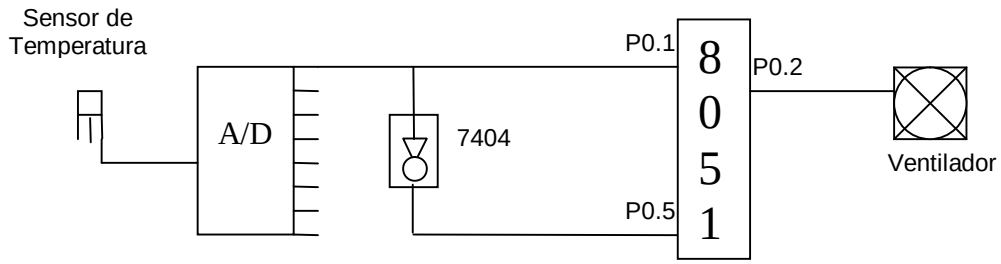


Figura 22. Diagrama em blocos do sistema de controle de temperatura

Neste sistema, o microcontrolador é utilizado para controlar a ligação ou desligamento de um ventilador de acordo com a temperatura do ambiente. O sensor de temperatura retorna uma determinada voltagem de acordo com a temperatura que se encontra. Quando a temperatura atinge 25º Celsius, a saída retorna 0,25V. Ao passar pelo conversor A/D, esta voltagem é alterada para um valor digital, o qual será trabalhado. Supõe-se que o valor binário retornado pelo conversor A/D seja 10000000b quando a temperatura for maior ou igual a 25º Celsius, e 00000000b quando for menor que 25º Celsius.

Portanto, o bit 0 define o aquecimento ou não do ambiente, sendo que quando estiver em nível lógico 1, o ambiente está com temperatura superior ou igual a 25º Celsius e o ventilador precisa ser ligado. Caso contrário, se o bit 0 estiver em nível lógico baixo, o ambiente estará com temperatura menor que 25º Celsius e o ventilador deverá ser desligado.

O sistema a ser executado dentro do microcontrolador é descrito por meio de um modelo em Rede de Petri Lugar/Transição, cujo grafo é mostrado na Figura 23. O lugar **`input(P0.1)`** é um lugar de controle, cuja entrada vem do pino P0.1 do microcontrolador

8051 da Intel. O lugar **`input(P0.5)`** é um lugar de controle, cuja entrada vem do pino P0.5. Como tem um inversor (7404) ligado na saída do conversor A/D paralelo (flash), onde o pino P0.1 recebe a saída sem modificação do conversor A/D e o pino P0.5 recebe o valor invertido do sinal, os valores destes pinos sempre serão inversos, ou seja, apenas estas duas combinações de valores serão possíveis: 0 para P0.1 e 1 para P0.5 ou 1 para P0.1. e 0 para P0.5.

Quando os valores forem 1 e 0, nos pinos P0.1 e P0.5 respectivamente, significa que o sensor atingiu uma faixa de temperatura alta, e que o sistema precisa ligar o ventilador. Caso contrário, quando os valores forem 0 e 1, nos pinos P0.1 e P0.5 respectivamente, o ventilador será desligado e o sistema ficará ocioso até que a temperatura aumente novamente.

O sistema modelado em Rede de Petri Lugar/Transição dado na Figura 23 possui inicialmente uma marca no lugar P0 e uma marca no lugar P1. É importante ressaltar que estes dois lugares não representam nenhuma porta do microcontrolador e, para que fossem associadas às portas do microcontrolador, seria necessário escrever as palavras-chaves ***input*** ou ***output*** nos rótulos dos lugares.

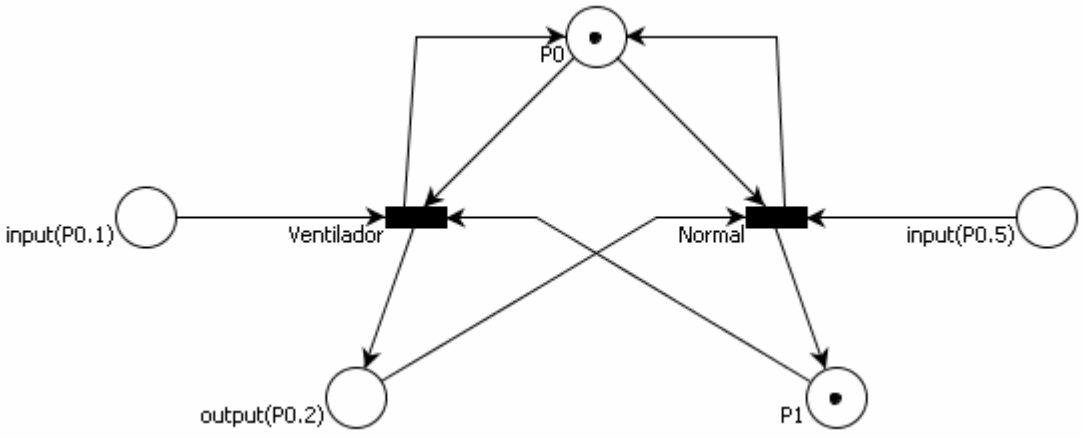


Figura 23. Modelo de Rede de Petri de um sistema de controle de temperatura

Neste estado, o sistema permanece ocioso até que se tenha uma marca no sinal de controle referenciada pela porta P0.1. Quando um sinal de controle em nível lógico alto for dado no pino P0.1, a transição “Ventilador” estará habilitada. Quando esta transição for

disparada, ele removerá a marca do lugar P0 e a marca do lugar P1 e colocará uma nova marca no lugar simbolizado como um sinal de controle da porta P0.2. Quando o pino P0.2 receber o sinal de nível lógico alto, ele ativará o ventilador.

O sistema permanecerá ocioso até o momento em que o bit enviado para a porta P0.1 vier a ser 0 e conseqüentemente o bit enviado para a porta P0.5 vier a ser 1. Quando isso ocorrer, as marcas do lugar P0 e do lugar de controle output(P0.2) serão removidas e uma nova marca será criada em P1. A partir do momento em que o sinal de controle for removido, ou seja, o momento no qual o sistema envia um bit de nível lógico baixo para a porta P0.2, o ventilador será, conseqüentemente, desligado.

É importante ressaltar que os sinais de controle de entrada, reconhecidos pela palavra-chave ***input***, permanecem nos seus respectivos lugares até que o bit enviado para tal porta vá para o nível lógico baixo e nunca serão alterados pelo programa. Outra consideração importante é em relação à quantidade de marcas nos lugares de controle, que devem conter apenas 1 ou 0 marcas já que estes lugares são diretamente ligados aos pinos das portas do microcontrolador 8051 da Intel.

O código gerado para esta rede de Petri, por meio da ferramenta de geração automática de código, é dado na Figura 24. Em seguida, são explicados o código *assembly* gerado pela ferramenta.

Nesta rede, os lugares P0 e P1 são inicialmente marcados com uma marca (linha 01 e 02). Quando estes dois lugares contiverem uma marca cada, o sistema estará ocioso e com o ventilador desligado.

A transição Normal (linha 03 a 17) é responsável por desligar o ventilador e será disparada apenas quando houver uma marca no lugar P0, um sinal de controle, em nível lógico alto, na porta P0.5 e um sinal de controle, em nível lógico alto, na porta P0.2. A porta P0.2 será utilizada como porta de saída e portanto seu valor pode ser alterado por *software*.

Para isso, ocorre a verificação do valor na porta P0.2 (linha 03), onde o valor da porta é armazenado no Carry Bit e em seguida é invertido (linha 04), através da instrução `CPL`. Na linha 05 há uma instrução `JC`, a qual realiza um salto para o endereço de memória de programa, dado pelo rótulo “ET1F”, caso o valor do *Carry Bit* seja um, isto é, se o pino 2 da porta P0 estiver em nível lógico alto.

```
01    MOV 14h,#1h          ;lugar: P0
02    MOV 15h,#1h          ;lugar: P1

03    ET1I:  MOV C,P0.2    ;transição: Normal
04           CPL C
05           JC ET1F
06           MOV A,14h
07           CLR C
08           SUBB A,#1h
09           JC ET1F
10           MOV C,P0.5
11           CPL C
12           JC ET1F
13           CLR P0.2
14           DEC 14h
15           INC 15h
16           INC 14h
17    ET1F:  LJMP ET0I

18    ET0I:  MOV A,14h     ;transição: Ventilador
19           CLR C
20           SUBB A,#1h
21           JC ET0F
22           MOV C,P0.1
23           CPL C
24           JC ET0F
25           MOV A,15h
26           CLR C
27           SUBB A,#1h
28           JC ET0F
29           DEC 14h
30           DEC 15h
31           SETB P0.2
32           INC 14h
33    ET0F:  LJMP ET1I

34           END
```

Figura 24. Código Assembly referente ao modelo de Rede de Petri da Figura 23

Entre as linhas 06 e 09 é verificado se há uma marca em P0 (14h), por meio da instrução `SUBB`. Quando se executa esta transição, o valor do *Carry Bit* é alterado para zero se o primeiro operando (A) é maior ou igual ao segundo operando (1h) e, caso contrário, é

alterado para um. Com isso, pode-se determinar se há a quantidade de marcas necessária nos lugares de entrada, dado pelo peso do arco.

As linhas 10 e 11 são responsáveis por verificar se o pino de controle 5 da porta P0 está em nível lógico 1 e, se estiver, a transição é dispara (linha 13 a 16). Na linha 13, o pino 2 da porta P0 é alterado para nível lógico baixo. Na linha 14, a marca do lugar P0 é removida. Na linha 15 a marca do lugar P1 é removido e na linha 16 uma nova marca é criada no lugar P0.

A transição Ventilador (linha 18 a 33) é responsável por ligar o ventilador e será disparada apenas quando houver uma marca no lugar P0, uma marca no lugar P1 e um sinal de controle, em nível lógico alto, na porta P0.1. Para ser disparada, entre as linhas 18 e 21 é verificado se há uma marca em P0 (14h), por meio da instrução `SUBB`. Quando se executa o código desta transição, o valor do *Carry Bit* é alterado para zero se o primeiro operando (A) é maior ou igual ao segundo operando (1h) e, caso contrário, é alterado para um. Com isso, pode-se determinar se há a quantidade de marcas necessária nos lugares de entrada, dado pelo peso do arco.

A linha 22 é responsável por verificar se há um sinal em nível lógico 1 no pino 1 da porta P0, onde o valor da porta é armazenado no Carry Bit e em seguida é invertido (linha 23), através da instrução `CPL`. Na linha 24 há uma instrução `JC`, a qual realiza um salto para o endereço de memória de programa, dado pelo rótulo “ET0F”, caso o valor do *Carry Bit* seja um, isto é, se o pino 1 da porta P0 estiver em nível lógico alto.

As linhas 25 a 28 são responsáveis por verificar se há uma marca no lugar P1. Caso não haja, o programa salta para a linha 33, a qual, por meio da instrução `LJMP`, realiza um salto para o início da transição Normal.

Caso a transição Ventilador esteja habilitada para ser disparada, a linha 29 remove uma marca do lugar P0 (14h), a linha 30 remove uma marca do lugar P1 (15h), a linha 31

altera para nível lógico alto o pino 2 da porta P0, o que, consequentemente, liga o ventilador. Por fim, a linha 32 cria uma nova marca no lugar P0 (14h).

Por meio dos testes realizados neste capítulo, pode-se concluir que o código gerado corresponde exatamente ao modelo de rede de Petri correspondente. Além disso, todos os códigos gerados neste capítulo foram simulados nos programas PEQui [PEQUI, 2005] e 535 Simulator [535 Simulator, 2005] e executaram exatamente o que foi proposto pelas redes de Petri correspondentes.

CONCLUSÃO

Com o objetivo de automatizar o projeto de sistemas digitais, reduzindo o tempo gasto na transformação de uma especificação de um sistema em um projeto simulado e validado, criou-se o ambiente de co-projeto de *hardware* e *software*, o qual é dividido em várias etapas, incluindo a síntese de *software*, a qual é o foco deste trabalho e realiza a transformação de uma especificação refinada da parte correspondente ao *software* em uma linguagem de programação para uma determinada arquitetura-alvo.

O modelo formal de entrada para a etapa de síntese de *software* é a rede de Petri Lugar/Transição, a qual foi escolhida após comparações que mostraram que a transformação de um modelo de rede de Petri Colorida gera código *assembly* maior que a rede de Petri Lugar/Transição, mesmo tendo o modelo mais compacto.

Durante todo o processo de engenharia de *software* da ferramenta para geração automática de código utilizou-se o paradigma de Orientação a Objetos, o que permite a manutenção e reutilização das classes criadas de forma fácil e ágil.

Foi realizada neste trabalho uma comparação entre as linguagens orientadas a objetos C++ e Java, que mostrou o uso da linguagem Java, neste contexto de co-projeto, como sendo a linguagem mais apropriada para a codificação da ferramenta de geração de código. Por meio do uso desta linguagem, o programa possui grande portabilidade, além de poder ser executado em qualquer computador que execute a Máquina Virtual Java (JVM) e acoplado a ambientes de co-projeto compatíveis com programas feitos em Java e que usem redes de Petri Lugar/Transição como modelo interno.

O ambiente de co-projeto possui como arquitetura-alvo uma pequena variável de tipos de microprocessadores e microcontroladores, dentre os quais estão o microcontrolador 8051

da Intel, o qual é muito utilizado na implementação de sistemas de controle e foi utilizado como a arquitetura-alvo para a qual a ferramenta gera o código *assembly*.

Pode-se concluir que o código gerado representa exatamente os modelos de rede de Petri correspondentes, o que indica que o programa gera adequadamente o código *Assembly* desejado. Além disso, o código gerado pela ferramenta não precisa ser modificado para ser gravado e executado pelo microcontrolador, agilizando o processo de síntese do modelo em rede de Petri para uma arquitetura-alvo.

Em trabalhos futuros poderá ser estudada uma maneira de criar variantes do programa para a geração de código para outras arquiteturas que venham a ser acopladas nos ambientes de co-projeto de *hardware* e *software*.

REFERÊNCIAS

535 Simulator. Disponível em: <http://3w.mundivia.es/hvasquez/sim535>. Acesso em: 10 de outubro de 2005.

Artifex. Disponível em: <http://artis-software.com>. Acesso em: 25 de outubro de 2005.

Cardoso, J.; Valette, R. **Redes de Petri**. Florianópolis: Editora da Universidade Federal de Santa Catarina, 1997, 212p.

CPN-AMI. Disponível em: <http://www-src.lip6.fr/logiciels/mars/CPNAMI/>. Acesso em: 25 de outubro de 2005.

De Micheli, G; Sami, G. ***Hardware/Software Co-Design***. Dordrecht: Kluwer Academic Publishers, 1996, v.310, p.397-426.

Deitel H. M.; Deitel P. J. **Java: Como programar**. 4.ed. Porto Alegre: Bookman, 2003, 1386p.

Gimenez, S. P. **Microcontroladores 8051**. 1.ed. São Paulo: Prentice Hall, 2002, 253p.

Gupta, R.G. ***Co-Synthesis of Hardware and Software for Digital Embedded Systems***. 1993. 262f. Tese (Doutorado em Engenharia Elétrica) – Stanford University.

Intel Corporation. ***MCS51: 8-bit control-oriented microcontrollers***. Data Sheet, 1994, 21 pp.

Jensen, K. **Coloured Petri Nets: Basic Concepts, Analysis Methods and Practical Use**. 2.ed. Germany: Springer, 1997, v.1, 234p.

Lakos, C.; Keen, C. ***LOOPN++: A new language for object-oriented petri nets***. *In proceedings of Modelling and Simulation (European Simulation Multiconference)*, 1994, 369–374p.

Larman, C. **Utilizando UML e Padrões**. 2. ed. Porto Alegre: Bookman, 2004, 607p.

Mortensen, K.H. ***Automatic Code Generation from Coloured Petri Nets for an Access Control System***, *In Kurt Jensen (ed.): Second Workshop on Practical Use of Coloured Petri Nets and Design/CPN,* Aarhus, Denmark, October 1999, 11-15p.

Nicolosi, D.E.C. **Microcontrolador 8051 Detalhado**. São Paulo: Érica, 2000, 227p.

Niemann R. ***Hardware/Software Co-Design for Data Flow Dominated Embedded Systems***. Boston: Kluwer Academic Publishers, 1998, 223p.

PEQui. Disponível em: <http://www.geocities.com/sim8051/>. Acesso em: 10 de outubro de 2005.

PIPE. Disponível em: <http://pipe2.sourceforge.net>. Acesso em: 10 de setembro de 2005.

PNML. Disponível em: <http://www.informatik.hu-berlin.de/top/pnml/about.html>. Acesso em: 08 de outubro de 2005.

Pressman, R. S. **Engenharia de *Software***. 5. ed. Rio de Janeiro: McGraw-Hill, 2002, 843p.

Schildt, H. **C: Completo e Total**. 3.ed. São Paulo: Pearson Education do Brasil, 1996, 827p.

SML/NJ. Disponível em: <http://www.smlnj.org/>. Acesso em: 15 de outubro de 2005.

Tanenbaum, A. S. ***Modern Operating Systems***. 2. ed. New Jersey: Prentice Hall, 2001, 960p.

Tanenbaum, A. S. ***Structured Computer Organization.*** Tradução de Nery Machado Filho. 4.ed. Rio de Janeiro: LTC, 1999, 398p.

Wolf, W; Staunstrup, J. ***Hardware/software co-design: Principles and Practice***. Boston: Kluwer Academic Publisher, 1997, 295p.

Wolf, W. ***A Decade of Hardware/Software Codesign***. IEEE *Computer Magazine*, 2003, v.36, n°.4, p.38-43.

ANEXO I

```
MOV 21h,#70h      ;ASCII p = 70h
MOV 22h,#00h

MOV 23h,#71h      ;ASCII q = 71h
MOV 24h,#00h

MOV 25h,#6Ah      ;ASCII j = 6Ah
MOV 26h,#00h

SETB 00h    ;hashi1
SETB 01h    ;hashi2
SETB 02h    ;hashi3

PROCH:MOV 21h,23h
      MOV 22h,24h
      MOV 23h,25h
      MOV 24h,26h
      MOV 25h,#00h
      MOV 26h,#00h
      RET

T0:   MOV 15h,21h
      MOV 16h,22h
      ACALL PROCH
      MOV A,15h
      CJNE A,#70h,T0p
      CJNE A,#71h,T0q
      CJNE A,#6Ah,T0j
      MOV 25h,15h
      MOV 26h,16h
      SJMP T1

T0p:  JNB 00h, T1
      JNB 01h, T1
      CLR 00h
      CLR 01h
      ACALL PROCH
      MOV 30h,15h
      MOV 31h,16h
      SJMP T1

T0q:  JNB 01h, T1
      JNB 02h, T1
      CLR 01h
      CLR 02h
      ACALL PROCH
      MOV 30h,15h
      MOV 31h,16h
      SJMP T1

T0j:  JNB 00h, T1
      JNB 02h, T1
      CLR 00h
```

```
        CLR 02h
        ACALL PROCH
        MOV 30h,15h
        MOV 31h,16h
        SJMP T1

T1:     MOV A,30h
        JNZ T2
        MOV 15h,30h
        MOV 16h,31h
        MOV A,15h
        CJNE A,#70h,T1p
        CJNE A,#71h,T1q
        CJNE A,#6Ah,T1j
        SJMP T2

T1p:    SETB 00h
        SETB 01h
        MOV 32h,15h
        MOV 33h,16h
        MOV 30h,#00h
        MOV 31h,#00h
        SJMP T2

T1q:    SETB 01h
        SETB 02h
        MOV 32h,15h
        MOV 33h,16h
        MOV 30h,#00h
        MOV 31h,#00h
        SJMP T2

T1j:    SETB 00h
        SETB 03h
        MOV 32h,15h
        MOV 33h,16h
        MOV 30h,#00h
        MOV 31h,#00h
        SJMP T2

T2:     MOV A,32h
        JNZ T0
        MOV 15h,32h
        MOV 16h,33h
        INC 16h
        MOV 25h,15h
        MOV 26h,16h
        MOV 32h,#00h
        MOV 33h,#00h
        SJMP T0

        END
```