Valério Gutemberg de Medeiros Junior

Aplicação do Método B ao Projeto Formal de Software Embarcado

Natal – RN

Setembro / 2009

Valério Gutemberg de Medeiros Junior

Aplicação do Método B ao Projeto Formal de Software Embarcado

Dissertação de mestrado apresentada ao Programa de Pós-graduação em Sistemas e Computação do Departamento de Informática e Matemática Aplicada da Universidade Federal do Rio Grande do Norte, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador:
Prof. Dr. David Boris Paul Déharbe

Universidade Federal do Rio Grande do Norte
Centro de Ciências Exatas e da Terra
Departamento de Informática e Matemática Aplicada
Programa de Pós-graduação em Sistemas e Computação

Natal/RN

Setembro / 2009

Dissertação de Mestrado sob o título *"Aplicação do Método B ao Projeto Formal de Software Embarcado"*, defendida por Valério Gutemberg de Medeiros Jr em 9 de Setembro de 2009, em Natal, Rio Grande do Norte, pela banca examinadora constituída pelos professores:

Prof. Dr. David Boris Paul Déharbe
Departamento de Informática e Matemática Aplicada - UFRN
Orientador

Prof. Dr. Anamaria Martins Moreira
Departamento de Informática e Matemática Aplicada - UFRN

Prof. Dr. Ana Cavalcanti
Departamento de Ciência da Computação da Universidade de York

---

Prof. Dr. André Laurindo Maitelli
Departamento de Engenharia de Computação e Automeação - UFRN

Dedico este trabalho primeiramente a Deus, por me dar toda capacidade de superar as inúmeras barreiras. A minha família, especialmente meus pais, por sempre me apoiar em todos os momentos com palavras e atos. E aos professores que contribuíram para minha formação acadêmica.

Agradecimentos

Dedico meus sinceros agradecimentos para:

- Deus por iluminar meu caminho e me dar forças para seguir sempre em frente;

- Os meus pais por todo apoio, incentivo e financiamento dos meus estudos;

- Os professores David Déharbe e Anamaria Martins que me orientaram e sempre que estavam disponíveis atendiam-me prontamente;

- A Rossana por compreender e ajudar-me nos longos períodos de estudo;

- Os professores do DIMAp que contribuíram para formação de um modo geral;

- Os meus familiares por todo apoio;

- Finalmente, todos os meus amigos e colegas de turma que indiretamente ajudaram-me nesse perído de estudo intenso.

"As dificuldades são como as montanhas.
Elas só se aplainam quando avançamos sobre elas"
Provérbio japonês

Resumo

Este trabalho apresenta um método de projeto proposta para verificação formal do modelo funcional do *software* até o nível da linguagem *assembly*. Esse método é fundamentada no método B, o qual foi desenvolvido com o apoio e interesse da multinacional do setor de petróleo e gás *British Petroleum* (BP). A evolução dessa metodologia tem como objetivo contribuir na resposta de um importante problema, que pertence aos grandes desafios da computação, conhecido como "*The Verifying Compiler*". Nesse contexto, o presente trabalho descreve um modelo formal do microcontrolador *Z80* e um sistema real da área de petróleo. O modelo formal do *Z80* foi desenvolvido e documentado, por ser um pré-requisito para a verificação até nível de *assembly*. A fim de validar e desenvolver a metodologia citada, ela foi aplicada em um sistema de teste de produção de poços de petróleo, o qual é apresentado neste trabalho. Atualmente, algumas atividades são realizadas manualmente. No entanto, uma parte significativa dessas atividades pode ser automatizada através de um compilador específico. Para esse fim, a modelagem formal do microcontrolador e a modelagem do sistema de teste de produção fornecem conhecimentos e experiências importantes para o projeto de um novo compilador. Em suma, esse trabalho deve melhorar a viabilidade de um dos mais rigorosos critérios de verificação formal: acelerando o processo de verificação, reduzindo o tempo de projeto e aumentando a qualidade e confiança do produto de *software* final. Todas essas qualidades são bastante relevantes para sistemas que envolvem sérios riscos ou exigem alta confiança, os quais são muito comuns na indústria do petróleo.

Palavras Chaves: Engenharia de Software, Métodos Formais, Verificação de *Assembly*, Método B.

Abstract

This work shows a project method proposed to design and build software components from the software functional model up to assembly code level in a rigorous fashion. This method is based on the B method, which was developed with support and interest of *British Petroleum* (BP). One goal of this methodology is to contribute to solve an important problem, known as "*The Verifying Compiler*". Besides, this work describes a formal model of *Z80* microcontroller and a real system of petroleum area. To achieve this goal, the formal model of *Z80* was developed and documented, as it is one key component for the verification upto the assembly level. In order to improve the mentioned methodology, it was applied on a petroleum production test system, which is presented in this work. Part of this technique is performed manually. However, almost of these activities can be automated by a specific compiler. To build such compiler, the formal modelling of microcontroller and modelling of production test system should provide relevant knowledge and experiences to the design of a new compiler. In summary, this work should improve the viability of one of the most stringent criteria for formal verification: speeding up the verification process, reducing design time and increasing the quality and reliability of the product of the final software. All these qualities are very important for systems that involve serious risks or in need of a high confidence, which is very common in the petroleum industry.

Keywords: Software Engineering, Formal Methods , Verified Compilation, B method.

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

É cada vez mais comum encontrar *software* embutido nos mais diversos produtos. Esse é um tipo *software* de propósito especial implementado em um computador simplificado e projetado para executar uma ou poucas funções dedicadas. Eles são encontrados em uma vasta gama de projetos: controles em navios, controles em aviões, injeção eletrônica em carros, elevadores e em vários dispositivos médicos [Gajski e Vahid 1995]. Atualmente a complexidade e o tamanho desse tipo de *software* vêm crescendo juntamente com o seu fator crítico. Com o uso em larga escala desse tipo de *software*, muitas vezes usado no controle de dispositivos de segurança, surgiu a necessidade de oferecer uma garantia mais rigorosa em relação ao seu correto funcionamento. Para prover uma garantia estabelecida matematicamente, técnicas de especificação e verificação formal são utilizadas.

No desenvolvimento de *softwares* críticos e embarcados, o processo de especificação pode ser bastante custoso e demorado. Porém o tempo de projeto de um sistema é um fator comercial muito importante devido a várias razões. Uma das principais é que os produtos pioneiros têm suas vendas consideravelmente maiores e com uma maior margem de lucro com relação aos produtos concorrentes lançados posteriormente.

No entanto, existem no mercado certificados de níveis de segurança, que valorizam a verificação formal desses produtos, o que em aplicações críticas é fundamental para a qualidade do produto [Dondossola 1999] . Segundo [Cury 2007], o processo de certificação de *software* tem como objetivo principal verificar ou validar a implementação dos requisitos de *software*, sendo que, atualmente, a técnica mais confiável para realização desse processo é baseada em argumentos com fundamentos matemáticos.

Para satisfazer as necessidades de tempo de projeto, segurança e robustez, técnicas fundamentadas matematicamente são utilizadas, o que também fornece um esquema teórico rigoroso no desenvolvimento de *software*. Essas técnicas são chamadas de métodos formais. Tais métodos podem acelerar o processo de especificação, pois facilitam o desenvolvimento de sistemas confiáveis através de ferramentas e técnicas de auxílio a

especificação, verificação e às vezes até geração de código. Portanto, esses métodos são adequados para agilizar o processo de projeto e validação de sistemas críticos.

Existem várias linguagens e metodologias formais, contudo este trabalho adota o método B [Abrial 1996]. Esse método foi desenvolvido com o apoio e interesse da multinacional do setor de petróleo e gás *British Petroleum* (BP). O método B tem se mostrado significativamente maduro e vem sendo usado no desenvolvimento de diversos sistemas de segurança crítica na Europa. Ele é um processo abrangente de desenvolvimento de *software* que suporta a especificação, verificação, refinamento e geração de código. Mas também, ele é particularmente forte na noção de desenvolvimento em níveis, o que permite o desenvolvimento de um sistema relativamente complexo ser decomposto em uma composição de módulos usando um pequeno número de construções [B-Core 1999]. Além disso, existem várias ferramentas de suporte a B [Clearsy 2009, Abrial 2007, B-Core 1995]. O AtelierB [Clearsy 2009] é a sua principal ferramenta; ele suporta vários recursos avançados e interessantes: provador interativo, provador distribuído em rede, gerador de código em linguagem de programação, etc. As potencialidades do método B tornam mais viáveis as especificações e certificações de sistemas de escala industrial. Então, por essas e outras capacidades, o interesse na indústria por B é crescente.

Uma outra razão para escolha do método B é que outros trabalhos [Aljer et al. 2003, Leuschel 2008, Ludovic e Lanet 1999, Evans e Grant 2008] com objetivo semelhante ao do presente trabalho também utilizam B. Adicionalmente, para o desenvolvimento de *software*, o método B é mais adequado que o Event-B, pois o método B possui construções mais próximas das linguagens de programação e suas ferramentas são capazes de gerar código nessa linguagem. Além disso, o suporte ferramental para B era mais robusto quando o trabalho foi iniciado.

O método B possui uma capacidade especial: cada passo da especificação B pode adicionar mais detalhes em diferentes níveis de abstração. Em B esse passo é denominado refinamento. O refinamento é uma técnica usada para verificar a transformação de um modelo abstrato de *software* (especificação) em um outro modelo matemático mais concreto [Abrial 1996, Morgan 1989]. Tal técnica permite que o método B suporte a verificação em um nível suficientemente próximo do nível de linguagem de programação estruturada. Enfim, a capacidade de especificar modelos de *software* e verificar a correspondência entre o modelo e a implementação merece destaque, porque isso é um requisito importante para a verificação de programas.

Com um modelo de especificação formal próximo do nível de linguagem de progra-

mação, muitas vezes, as próprias ferramentas de especificação são capazes de transformar a especificação em código de uma linguagem de programação como, por exemplo, C. Porém, em geral, não existe uma garantia formal da consistência da geração de código *assembly* a partir do modelo especificado ou código em uma linguagem de programação. Nesse processo de geração de código o compilador/gerador pode também introduzir erros. Para resolver esse problema de forma confiável, existe a proposta apresentada em [Dantas et al. 2008] para verificar formalmente a consistência do refinamento entre o modelo algorítmico especificado e o código sintetizado (*assembly* gerado).

Um desenvolvimento formal com o método B de um sistema simples foi realizado em equipe até o nível de *assembly* em três diferentes plataformas: Z80 (autor desta dissertação), 8051 (Stephenson Galvão) e PIC (David Déharbe). A descrição desse experimento encontra-se em [Dantas et al. 2008].

A especificação da linguagem *assembly* das três diferentes plataformas PIC, A8051 e Z80 possuíam conceitos comuns, motivando uma padronização desses conceitos. Além disso, existia a necessidade de concluir a especificação completa de uma plataforma, desenvolver estudos de caso mais avançados e aplicar novas e diferentes técnicas de verificação.

Para essas necessidades, o presente trabalho descreve o desenvolvimento da especificação de duas bibliotecas para padronizar os conceitos comuns das diferentes plataformas; a construção do modelo formal de um microcontrolador com a representação de todas as instruções e ações que modificam seu estado e um estudo de caso simples de um *software* embarcado verificado até o nível de *assembly*. Além disso, esse trabalho detalha as diferentes técnicas de verificação utilizadas, o que deve colaborar bastante nos trabalhos futuros. Em suma, esse trabalho tem como principal objetivo prover meios de verificar *software* em nível de *assembly* e apresentar as experiências e técnicas utilizadas para isso.

Nesse contexto, esse trabalho apresenta a modelagem B do microcontrolador Z80, a qual é usada para aplicar a solução citada em um sistema de teste de produção de poços de petróleo. Dessa forma, as experiências obtidas nessa atividade contribuirão indiretamente para o projeto de uma ferramenta com objetivo de automatizar o desenvolvimento verificado provido pelo método até o nível de *assembly*.

1.1 Motivação

Com o crescente aumento da complexidade do *software* e da necessidade cada vez maior de confiança em *software* embarcado, faz-se necessário prover uma forma de garantir

cada vez mais precisamente o correto funcionamento dos programas em suas plataformas de execução. Essa necessidade ocorre principalmente quando não se tem uma garantia formal, nos compiladores tradicionais, sobre a consistência da transformação do código fonte em relação a seu programa *assembly* gerado. Em geral, o processo de compilação pode introduzir erros sutis nos programas gerados. Então, um dos grandes desafios da computação é o desenvolvimento de uma ferramenta verificadora de compilação. Essa ferramenta usa matemática e lógica para verificar formalmente a correção dos programas que ela compila [Hoare 2005]. Assim, esse desafio está diretamente relacionado com esse trabalho, pois as atividades do presente trabalho são passos iniciais para construção dessa ferramenta de compilação.

Existem vários exemplos de desastres em que o uso intenso da verificação formal era teoricamente indispensável. Um deles foi o do foguete Ariane 5[1] em 1996 que explodiu 37 segundos após o lançamento. Esse foguete carregava quatro satélites cujo custo era U$ 500 milhões. O problema foi provocado por um erro de projeto, na tentativa de converter um número de 64 bits de ponto flutuante para um inteiro de 16 bits, ocorrendo um transbordamento. Isso provocou a emissão de sinais incorretos aos motores e implicou o acionamento da autodestruição do foguete. É importante destacar que a linguagem de programação utilizada era Ada, uma linguagem que possui os mais altos certificados de segurança.

É muito comum *softwares* embarcados da área de petróleo e gás envolverem complexidade, riscos e exigir alta confiança. Dessa forma, um meio de oferecer maiores garantias sobre o correto funcionamento desse tipo de *software* é aplicando a metodologia proposta em [Dantas et al. 2008] em um sistema real da área, o qual é apresentado na seção 5.2. Enfim, a evolução dessa metodologia pode render em longo prazo elevados certificados de segurança a esse tipo de sistema.

A exigência de certificações de segurança nos compiladores mais populares, como o da linguagem C para a arquitetura 80x86[2], não é uma grande preocupação dos projetistas; porque esses compiladores são usados em larga escala. Assim, seu uso em massa faz o papel de um grande conjunto de testes. Além disso, esses compiladores não têm como foco questões de forte segurança, pois na maioria das vezes não são usados em aplicações críticas.

[1] Ariane 5 é um tipo de foguete projetado para colocar satélites artificiais em órbitas geoestacionárias e enviar cargas para órbitas de baixa altitude.

[2] 80x86 é nome genérico dado para a arquitetura da família dos processadores baseados no 8086, da Intel Corporation.

Os compiladores desenvolvidos para um nicho menor de aplicações não contam com os testes de uso em larga escala. E quando esses compiladores são utilizados na compilação de programa de aplicações críticas se faz de extrema necessidade que eles ofereçam critérios adequados de qualidade. Esse é comumente o caso dos compiladores utilizados para microcontroladores e microprocessadores de propósitos especiais. Portanto, essas plataformas estão no foco das pesquisas deste trabalho.

A metodologia proposta em [Dantas et al. 2008] é a adotada neste trabalho. Ela ainda tem custos de aplicação relativamente altos, contudo esses podem reduzir significativamente. Esses custos podem diminuir com a implementação de um compilador para a metodologia proposta e com o desenvolvimento de novas técnicas. Para satisfazer essas necessidades pode-se reutilizar alguns componentes de *software* de suporte ao método B e também definir uma estratégia de prova mais específica para a verificação no nível de *assembly*, o que deve diminuir sensivelmente o custo do processo. Enfim, pesquisas intensificadas nesse sentido podem render ótimos resultados.

1.2 Trabalhos relacionados

Atualmente, os métodos utilizados para tentar garantir a consistência da transformação entre um *software* em determinada linguagem de programação e o código *assembly* correspondente, para uma determinada plataforma, são baseados em simulações, testes, compiladores certificados e verificação formal sobre o código gerado .

Para os testes serem realizados, cria-se um conjunto de programas exemplos, que contêm diversas combinações de sequências de instruções, as quais possivelmente permitam identificar erros. Então, após a compilação desses programas, eles são executados e em seguida são comparados os resultados esperados com os obtidos. Com essa avaliação é possível localizar vários problemas, porém não se garante a coerência entre o programa fonte e o código sintetizado, e também a ausência de erros de: implementação e consistência com relação aos requisitos do sistema.

Na abordagem de compiladores certificados, os projetistas da linguagem Ada[3] tinham como objetivo criar uma linguagem segura para ser usada pelo Departamento de Defesa Americana, em aplicações críticas de sistemas embarcados. Para isso foi desenvolvido um conjunto de certificações, com o intuito de assegurar a confiança do compilador. Essas

[3]Ada é uma linguagem de programação criada através de um concurso realizado pelo Departamento Americano de Defesa (*DoD*), o seu principal projetista foi o cientista Jean Ichbiah.

certificações de compilação continham testes próprios de Ada, que antecediam o primeiro padrão de teste de 1983 e é produto de 15 anos de pesquisa sobre a conformidade de Ada. Segundo [Burkhardt 2000], o compilador Ada 95 foi testado continuamente e de forma muito exigente; um dos conjuntos de testes continha mais de 3600 testes de cobertura. Dessa forma, os projetistas da linguagem concluíram que nos testes de certificação de compilação crítica obtiveram sucesso [Burkhardt 2000]. Com o decorrer dos trabalhos de certificação, o compilador Ada obteve um alto grau de amadurecimento permitindo sua utilização em sistemas críticos. No entanto, apesar de ser uma boa estratégia, projeto de compiladores como esses tem enorme custo de desenvolvimento.

Existem na comunidade científica vários projetos e ferramentas (Exemplos: [Leinenbach, Paul e Petrova 2005, Blazy, Dargaye e Leroy 2006, Leroy 2006]) , os quais garantem formalmente a transformação de um programa em uma linguagem específica para a linguagem *assembly* de determinada plataforma. Por outro lado, esses projetos não realizam a verificação dos requisitos e propriedades do modelo do *software*, o que é proposto e modelado neste trabalho.

Uma alternativa completa e possível é construir um tradutor verificado do código da linguagem de especificação B para a linguagem de algumas dessas ferramentas, por exemplo o compilador CompCert [Leroy 2006]. Dessa forma, seria possível realizar a verificação dos requisitos e propriedades do modelo do *software* em B. Essa seria uma tradução com "distância semântica" pequena e o próprio compilador CompCert garantiria a conversão do código de linguagem de programação para o código *assembly*. No entanto, para automatizar essa tradução, é necessária a formalização da semântica operacional da linguagem de especificação B em Coq [Bertot 2004] e provar as regras de tradução entre as linguagens.

Estrutura do documento: O capítulo 2 fornece os conceitos básicos do método B, o que é necessário para o entendimento da metodologia proposta e do modelo do microcontrolador. O capítulo 3 descreve uma visão geral da metodologia. Os dois capítulos seguintes 4 e 5 apresentam a modelagem do microcontrolador Z80 e a modelagem do estudo de caso. As conclusões e a sugestões de trabalhos futuros são apresentadas no último capítulo deste trabalho.

2 Método B

Para uma boa compreensão da metodologia deste trabalho, este capítulo introduz os principais conceitos do método B. Este capítulo apresenta: algumas definições envolvidas, os diversos módulos de especificação e os seus elementos, as regras de substituição e as construções das obrigações de prova.

B é um método para modelagem modular, no qual cada módulo especifica um objeto de *software* em um determinado nível de abstração. Estes níveis seguem respectivamente do mais abstrato para o mais concreto: MACHINE, REFINEMENT e IMPLEMENTATION. Esses termos são convencionados neste trabalho respectivamente como: modelo funcional, refinamento ou modelo refinado, e modelo algorítmico. A seguir são mostrados na Figura 1 os passos tradicionalmente utilizados no método B.

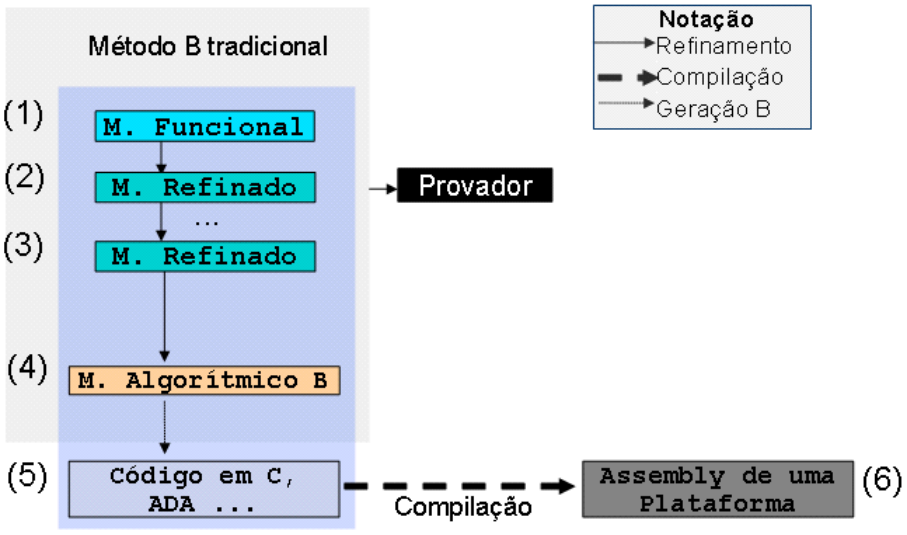

Figura 1: Etapas de verificação B tradicional.

Os níveis desse processo estão numerados de acordo com a ordem de desenvolvimento do *software*. Portanto, inicia a partir de um modelo funcional (1), realiza-se opcionalmente uma série de refinamentos sucessivos (2 - 3), constrói-se o modelo algorítmico (4), gera-se automaticamente o código fonte para uma determinada linguagem de programação (5) e

finaliza-se com a compilação do programa final (6).

O modelo de um sistema ainda pode ser composto de vários módulos interdependentes, ou seja, também suporta uma composição de modelos. Três cláusulas de composição serão descritas. A cláusula *INCLUDES* pertence à notação do modelo funcional e permite incluir instâncias de outros modelos funcionais [Abrial 1996]. A cláusula *SEES* referencia um modelo funcional e permite acessar suas constantes, funções e variáveis, porém sem modificá-las. A cláusula *IMPORTS* permite importar, em um modelo algorítmico, um modelo funcional.

Nas seções seguintes deste capítulo, serão apresentadas as características mais relevantes sobre os níveis de especificação B e as suas obrigações de prova.

2.1 Modelo funcional

Um modelo funcional é uma máquina de estado, na qual é descrito um conjunto de propriedades sobre os estados e operações que modificam o estado do modelo. O conjunto dos dados representados pelas variáveis constitui um estado do modelo. Esses dados são modelados por conceitos matemáticos, tais como conjuntos, relações, funções, sequências e árvores. A mudança de estado de um modelo é representada unicamente pelas operações, que por sua vez, fazem a alteração do valor de forma a preservar a integridade do modelo. A seguir é ilustrado um modelo funcional de exemplo.

MACHINE
  Caixa
VARIABLES
  saldo
INVARIANT
  $saldo \in \mathcal{Z} \wedge saldo >= 0$
INITIALISATION
  $saldo := 100$
OPERATIONS
  saque (*valor_saque*) =
  PRE
    $valor\_saque \in \mathcal{Z} \wedge$
    $valor\_saque \leq saldo$
  THEN
    $saldo := saldo$ - *valor_saque*
  END
END

Figura 2: Exemplo de um modelo funcional simples.

A Figura 2 contém um exemplo simples de especificação de um controle de crédito com a operação saque. Esse módulo contém as cláusulas básicas da composição de um modelo funcional. A primeira cláusula (**MACHINE**) descreve o nome do módulo, nesse caso "Caixa". A seguinte (**VARIABLES**) apenas declara o nome de suas variáveis. A terceira (**INVARIANT**) determina os tipos e as propriedades das variáveis. Portanto, essa cláusula requer que existam valores para a lista de variáveis do modelo a fim de que o (**INVARIANT**) seja válido, em outras palavras, que ela não seja inconsistente [Schneider 2001].

A próxima (**INITIALISATION**) define o estado inicial das variáveis do modelo. Isso significa que a execução da inicialização deve fazer com que o valor das variáveis satisfaça o **INVARIANT**. No modelo "Caixa", por exemplo, a inicialização da variável saldo com 100 estabelece o **INVARIANT**, pois 100 pertence ao conjunto dos inteiros positivos.

E por fim, a cláusula (**OPERATIONS**) define de que forma as operações modificam o valor das variáveis. As operações são especificadas através de construções chamadas substituições generalizadas. Então, para garantir que o modelo permaneça sempre em um estado válido, o método B define regras para construção de obrigações de prova. Nesse contexto as obrigações de prova são fórmulas lógicas que, quando válidas, garantem a permanência do modelo em estados válidos.

O módulo da Figura 2, por exemplo, contém a operação *saque*. Para garantir que essa operação não leva a um estado inválido, ela é restrita a saques de valores nunca maiores que o saldo atual. Essa restrição pode ser observada na pré-condição da operação. Assim, toda operação de mudança de valor dos dados deve preservar as condições de integridade estabelecidas no modelo, nesse caso, as propriedades estabelecidas no **INVARIANT**. Para tal, B oferece um conjunto de construções de obrigações de prova, que por sua vez são construídas através das regras de substituição.

A semântica das substituições é definida pelo cálculo das substituições, um sistema de reescrita dos diferentes tipos de substituições em fórmulas da lógica de primeira ordem. A Tabela 1 apresenta o nome das principais substituições generalizadas e as respectivas regras de reescrita.

Num primeiro momento, é importante entender a notação utilizada ($[S]P$) nas regras de substituição generalizada. A notação $[S]P$ significa que uma substituição S será aplicada em uma fórmula P. A seguir essa notação será exemplificada e ilustrada na expressão 2.1.

Nome	Regra de substituição
SKIP	**[skip]** $Q \equiv Q$
BEGIN	$[\textbf{BEGIN}\, G\, \textbf{END}]Q \equiv [G]Q$
SIMPLE	$[\alpha := \beta]Q \equiv Q'$
PRE	$[\textbf{PRE}\, P\, \textbf{THEN}\, S\, \textbf{END}]Q \equiv (P \wedge [S]Q)$
IF	$[\textbf{IF}\, P\, \textbf{THEN}\, S_1\, \textbf{ELSE}\, S_2\, \textbf{END}]Q \equiv (P \Rightarrow [S_1]Q) \wedge (\neg P \Rightarrow [S_2]Q)$
CASE	$[\textbf{CASE}\, E\, \textbf{OF EITHER}\, e_1\, \textbf{THEN}\, S_1\, \textbf{OR}\, e_2\, \textbf{THEN}\, S_2\, \textbf{END}]Q$ $\equiv (E = e_1 \Rightarrow [S_1]Q) \wedge (E = e_2 \Rightarrow [S_2]Q)$
ANY	$[\textbf{ANY}\, x\, \textbf{WHERE}\, P\, \textbf{THEN}\, S\, \textbf{END}]I$ $\equiv \forall x(P \Rightarrow [S]I)$
PARALLEL	$[x := E \| \| y := F]I \equiv [x, y := E, F]I$
SEQUENCING	$[S_1 \ ; \ S_2]Q \equiv [S_1][S_2]Q$

Tabela 1: Regras de substituição generalizada. Os símbolos utilizados nesta tabela estão definidos como: P, P_1, P_2, Q são predicados; S, S_1, S_2 são substituições; E uma expressão; e_1, e_2 são valores e α uma variável que é substituída por uma expressão β.

A primeira é a regra da substituição **SKIP**. Ela não modifica o estado das variáveis, contudo pode ser útil para especificar que alguma ramificação de **IF** não modifica o valor das variáveis, por exemplo.

A segunda **BEGIN** é usada para definir o escopo de uma substituição, ou seja, funciona como um "delimitador de bloco".

A terceira é a regra da substituição simples. O significado dessa regra é a substituição de uma variável α por uma expressão β em um termo Q, escrito como $[\alpha := \beta]Q$. Assim, se α for livre em Q a substituição acontece em toda ocorrência livre de α por β resultando em Q'. A expressão 2.1 ilustra um exemplo intuitivo do cálculo das substituições aplicando essa regra.

$$[v := w + v + z](w < v) \equiv (w < w + v + z) \tag{2.1}$$

A substituição **PRE** é usada para expressar os requisitos necessários para efetuar uma substituição, ou seja, para efetuar S é necessário garantir que P seja válido.

O **IF** é uma substituição condicional de comportamento determinístico. Essa é usada para definir um certo comportamento dependendo da validade do seu predicado condicional. O seu significado é o seguinte: se o predicado P é verdadeiro então efetue S_1,

caso contrário, efetue S_2. Um outro detalhe é que a substituição **ELSE** é opcional, assim como nas linguagens de programação.

A construção **CASE** é também uma substituição condicional de comportamento determinístico. No entanto, ela possui uma expressão E e uma lista de valores (e_1, e_2) para E. Assim, se o valor de E for igual a e_1 então efetue s_1, senão se o valor de E for igual a e_2 então efetue s_2. Essa construção tem semântica similar ao *switch* da linguagem C.

O **ANY** é uma substituição não determinística, em que X é uma lista de variáveis; P é um predicado com a tipagem e restrições sobre as variáveis de X; e S é uma substituição sobre as variáveis de X. Essa construção é muito comum e utilizada principalmente no primeiro nível de modelagem B.

A substituição **PARALLEL** é uma substituição representada por "||" e corresponde à execução das substituição de forma atômica. No caso particular do exemplo da Tabela 1, x e y recebem respectivamente, em um mesmo instante, o valor de E e F.

Finalmente, a **SEQUENCING** é uma substituição representada por "; " e corresponde à execução em sequência de várias substituições. Isso significa que o predicado obtido em $[S_1;S_2]Q$ é o mesmo obtido em $([S_1]([S_2]Q))$.

Obrigações de prova do modelo funcional: Como já dito, as obrigações de prova garantem a consistência do modelo e nesse trabalho todas as construções de obrigações de prova são fundamentadas na notação de [Abrial 1996]. As estruturas de dois modelos funcionais, um M_1 que importa um outro M, são ilustradas na Figura 3 com o intuito de estabelecer a notação.

Na construção das obrigações de prova do modelo funcional da Figura 3, algumas das expressões referentes ao modelo M_1 são numeradas com *1* e essa numeração é apresentada de forma subscrita. Entretanto, as expressões que não contêm a numeração subscrita pertencem ao modelo funcional M, o qual é incluído pelo modelo M_1.

As obrigações de prova para o modelo funcional da Figura 3 têm uma notação própria. Essa notação relaciona as cláusulas do modelo com os termos utilizados nas obrigações de prova. Cada modelo funcional contém fundamentalmente três tipos de construções de obrigações de prova. Essas incluem expressões comuns relativas aos diferentes elementos do modelo, como parâmetros, constantes, conjuntos e variáveis. As expressões indexadas por 1 são relativas ao modelo M_1. As obrigações de prova apresentadas nessa seção também se referem ao modelo funcional M_1 e suas expressões relacionadas são:

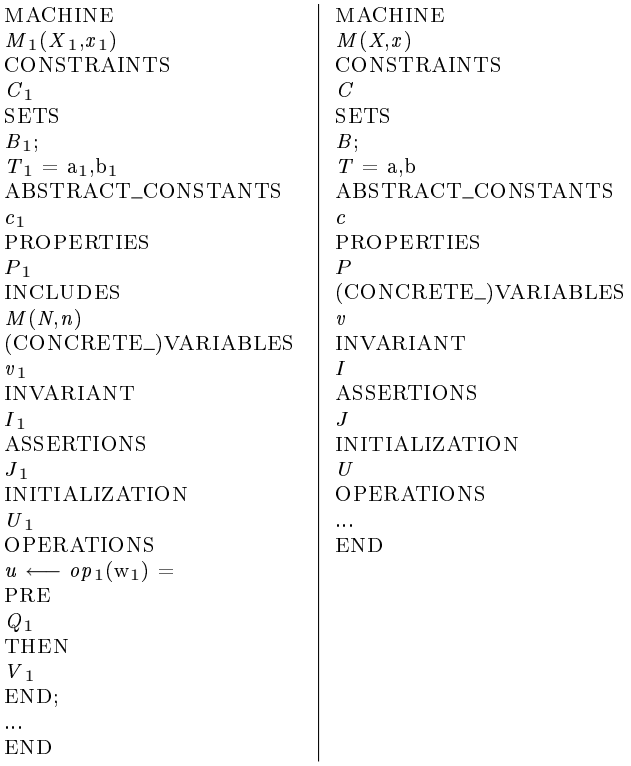

Figura 3: Estrutura modular do modelo funcional [Abrial 1996].

- A_1 - especifica o tipo dos conjuntos parâmetros do modelo. Por convenção, são conjuntos de inteiros não vazios. Assim, se X é o único conjunto parâmetro do modelo, A_1 é $X \in \mathbb{P}_1(\mathbb{Z})$;
- B_1 - especifica o tipo dos conjuntos declarados na cláusula **SETS**;
- C_1 - especifica as restrições sobre os parâmetros. É definido na cláusula **CONSTRAINTS**;
- P_1 - especifica as restrições sobre as constantes. É definido na cláusula **PROPERTIES**;
- I_1 - especifica as restrições sobre as variáveis de estado. É definido na cláusula **INVARIANT**;
- J_1 - especifica os lemas definidos na cláusula **ASSERTIONS**;
- α_1 - abrevia a expressão $A_1 \wedge B_1 \wedge C_1 \wedge P_1$;
- β_1 - abrevia a expressão $I_1 \wedge J_1 \wedge Q_1$, em que Q_1 é o predicado da pré-condição para cada operação.

A seguir são apresentadas as principais obrigações de prova do modelo funcional.

A primeira obrigação de prova do modelo funcional preocupa-se em garantir a correção em relação a inclusão de outros modelos funcionais. De acordo com a Figura 3 temos a inclusão do modelo funcional $M(X,x)$ sendo instanciado na forma $M(N,n)$. Nesse sentido a obrigação de prova deve instanciar os parâmetros do modelo com os parâmetros efetivos e verificar se satisfaz as restrições. A construção dessa obrigação de prova completa é representada na expressão 2.2.

$$\boxed{\alpha_1 \Rightarrow [X, x := N, n](A \wedge C)} \tag{2.2}$$

A segunda obrigação de prova verifica se a inicialização U_1 do modelo funcional M_1 e U do modelo importado M satisfazem as restrições e propriedades impostas em I_1 de M_1. Essa obrigação de prova é ilustrada na expressão 2.3.

$$\boxed{\alpha_1 \wedge B \wedge P \Rightarrow [[X, x := N, n]U][U_1]I_1} \tag{2.3}$$

Enfim, a última obrigação de prova, representada na expressão 2.4, garante que o corpo de cada operação preserva a cláusula **INVARIANT**. Na Figura 4 tem-se um exemplo prático e intuitivo da construção e resolução da obrigação de prova. Esse exemplo é um dos cálculos realizados para a verificação da operação do modelo funcional "Caixa". O primeiro passo foi expandir os elementos utilizados na construção da obrigação de prova. O passo seguinte foi aplicar as regras de substituição até resultar em um predicado. Então, esse predicado pode ser provado válido, o que significa que a operação preserva o invariante.

$$\boxed{\begin{array}{l}\alpha_1 \wedge \beta_1 \wedge B \wedge P \wedge [X, x := N, n](I \wedge J) \\ \Rightarrow [V_1]I_1\end{array}} \tag{2.4}$$

$$
\begin{array}{ll}
\alpha_1 \wedge \beta_1 \wedge B \wedge P & \equiv \quad I_1 \wedge Q_1 \Rightarrow [V_1]I_1 \\
\wedge\, [X, x := N, n](I \wedge J) \Rightarrow [V_1]I_1 & \\
& \equiv \quad I_1 \wedge Q_1 \Rightarrow [\textbf{PRE } Q_1 \textbf{ THEN } S \textbf{ END}]I_1 \\
& \equiv \quad I_1 \wedge Q_1 \Rightarrow Q_1 \wedge [S_1]I_1 \\
& \equiv \quad I_1 \wedge Q_1 \Rightarrow Q_1 \wedge [saldo := saldo - valor\_saque]I_1 \\
& \equiv \quad I_1 \wedge Q_1 \Rightarrow \\
& \qquad (Q_1 \wedge [saldo := saldo - valor\_saque]saldo \in \mathbb{Z} \\
& \qquad \wedge\, saldo >= 0) \\
& \equiv \quad I_1 \wedge Q_1 \Rightarrow (Q_1 \wedge (saldo - valor\_saque) \in \mathbb{Z} \\
& \qquad \wedge\, saldo >= 0) \\
& \equiv \quad I_1 \wedge Q_1 \Rightarrow (valor\_saque \in \mathbb{Z} \wedge valor\_saque \leq saldo \\
& \qquad \wedge\, (saldo - valor\_saque) \in \mathbb{Z} \wedge saldo >= 0) \\
& \equiv \quad true
\end{array}
$$

I_1 igual à: saldo $\in \mathcal{Z} \wedge$ saldo $>= 0$
Q_1 igual à: valor_saque $\in \mathcal{Z} \wedge valor\_saque \leq saldo$
S igual à: **PRE** Q_1 **THEN** S **END**
S_1 igual à: saldo := saldo - valor_saque

Figura 4: Exemplo prático da aplicação de regras de substituições.

2.2 Modelo refinado ou refinamento

Esta seção demonstra o conceito de refinamento, a importância dele para esse trabalho, alguns conceitos envolvidos e as construções de obrigação de prova que garantem a consistência entre os modelos.

Na especificação de sistemas, quanto maior o porte ou a complexidade, maior será a necessidade de utilizar o conceito de refinamento. Isso porque é muito difícil e complexo especificar corretamente um sistema diretamente no seu nível real concreto, ou seja, no nível de implementação.

Para amenizar essa dificuldade o método B permite especificar um modelo funcional simplificado, sem muitos detalhes e de forma abstrata. Então, após verificação da correção desse modelo mais simples, inicia-se a especificação de um modelo mais rico em detalhes. Nos níveis iniciais de especificação é comum o uso de operações não determinísticas, pois não é necessário, por exemplo, determinar o fluxo exato de execução de um algoritmo, então esse tipo de operação é usado para descrever um algoritmo de uma forma mais abstrata. Logo, o nível de modelagem algorítmica é restrito a construções determinísticas, que são aquelas em que todo o comportamento é precisamente detalhado e determinado.

Nesse ponto, o método B usa o conceito de refinamento para estabelecer uma correspondência entre o modelo simples e o mais detalhado. Dessa forma, torna-se facilitado o trabalho da construção do modelo final, porque essa construção é realizada de forma gradual através da construção de partes consistentes com a especificação funcional. O refinamento pode garantir a correspondência semântica fazendo o uso das suas leis. Ele tem papel fundamental nesse trabalho, pois é com esse conceito que é estabelecido a correspondência semântica entre os modelos de *software*.

2.2.1 Obrigações de prova do modelo refinado

Em B o modelo refinado tem estrutura semelhante a de um modelo funcional. Na Figura 5 são ilustrados três módulos, dois do modelo funcional e um modelo refinado. Esses serão utilizados para definir a notação das construções de obrigações de prova.

MACHINE	REFINEMENT	MACHINE
$M_1(X_1,x_1)$	$M_n(X1,x1)$	$M(X,x)$
CONSTRAINTS	REFINES M_{n-1}	CONSTRAINTS
C_1	SETS	C
SETS	B_n;	SETS
B_1;	T_n = a_n,b_n	B;
T_1 = a_1,b_1	ABSTRACT_CONSTANTS	T = a,b
CONSTANTS	c_n	CONSTANTS
C_1	PROPERTIES	c
ABSTRACT_CONSTANTS	P_n	ABSTRACT_CONSTANTS
c'_1	INCLUDES	c
PROPERTIES	$M(N,n)$	PROPERTIES
P_1	(CONCRETE_)VARIABLES	P
(CONCRETE_)VARIABLES	v_n	(CONCRETE_)VARIABLES
v_1	INVARIANT	v
INVARIANT	I_n	INVARIANT
I_1	ASSERTIONS	I
ASSERTIONS	J_n	ASSERTIONS
J_1	INITIALIZATION	J
INITIALIZATION	U_n	INITIALIZATION
U_1	OPERATIONS	U
OPERATIONS	$u_1 \longleftarrow op_1(w_1)$ =	OPERATIONS
$u_1 \longleftarrow op_1(w_1)$ =	PRE Q_n	$u \longleftarrow op(w)$ =
PRE	THEN	PRE
Q_1	V_n	Q
THEN	END;	THEN
V_1	...	V
END;	END	END;
...		...
END		END

Figura 5: Estrutura modular do modelo funcional e refinado [Abrial 1996].

Essa notação relaciona as cláusulas dos modelos com os termos utilizados nas obrigações de prova. Cada modelo refinado contém fundamentalmente quatro tipos de obrigações de prova. Essas obrigações de prova incluem expressões comuns, a maioria delas são as mesmas do modelo funcional.

Na construção das obrigações de prova do modelo refinado e algorítmico, algumas das

expressões são indexadas de acordo com o seu nível de abstração, dessa forma o modelo funcional é indexado com 1, o refinamento seguinte com 2, assim sucessivamente até o n-ésimo que representa o último modelo; para essa seção o refinado, mas que pode ser também o algorítmico. Entretanto, as expressões que não contêm índice pertencem ao modelo funcional M, o qual é incluído pelo modelo refinado M_n.

α_n e β_n formam hipóteses do n-ésimo modelo refinado, as quais são redefinidas da seguinte forma:

- α_n abrevia a expressão $A_1 \wedge B_1 \wedge ... \wedge B_n \wedge C_1 \wedge P_1 \wedge ... \wedge P_n$;
- β_n abrevia a expressão $I_1 \wedge ... \wedge I_n \wedge J_1 \wedge ... \wedge J_{n-1}$

A seguir são apresentadas as principais obrigações de prova do modelo refinado.

A primeira obrigação de prova do modelo refinado preocupa-se em garantir a correção em relação à inclusão dos modelos iniciais. De acordo com a expressão 2.5 temos a inclusão do modelo funcional $M(X,x)$ sendo instanciado na forma $M(N,n)$. Nesse sentido a obrigação de prova deve atualizar os parâmetros do modelo com os instanciados e verificar se satisfaz as restrições. A construção de obrigação de prova completa é representada na expressão 2.5.

$$\boxed{\alpha_n \Rightarrow [X, x := N, n](A \wedge C)} \tag{2.5}$$

A segunda obrigação de prova permite verificar se a inicialização do modelo refinado satisfaz as suas restrições e propriedades. Além disso, permite verificar se o resultado é exatamente o mesmo ou no mínimo coerente com a inicialização do modelo mais abstrato, ou seja, essa obrigação de prova deve ser satisfeita para garantir a relação entre a inicialização das variáveis do modelo abstrato e o concreto, em outras palavras, o modelo concreto não pode contrariar a substituição do modelo abstrato. Um exemplo válido é um modelo abstrato com uma variável x que recebe um valor pertencente aos inteiros positivos e o modelo concreto recebe o valor 10. Essa obrigação de prova é ilustrada na expressão 2.6.

$$\boxed{\alpha_n \wedge B \wedge P \Rightarrow [[X, x := N, n]U; U_n]\neg[U_{n-1}]\neg I_n} \tag{2.6}$$

Finalmente, a terceira obrigação de prova (2.7) preocupa-se com a consistência do refinamento das operações em M_n. Essa é a mais importante para o contexto desse trabalho, porque ela permite garantir que o código gerado seja coerente com a especificação, o que será explicado no próximo capítulo. Da mesma forma que a da inicialização do modelo refinado, a obrigação de prova da operação possibilita verificar se a operação satisfaz as suas restrições e propriedades. Além disso, permite verificar se o resultado é exatamente o mesmo ou é no mínimo coerente com a operação do modelo mais abstrato.

$$\begin{array}{l} \alpha_n \wedge \beta \wedge J_n \wedge Q_1 \wedge ... \wedge Q_{n-1} \wedge B \wedge P \wedge [X, x := N, n](I \wedge J) \\ \Rightarrow Q_n \wedge [[u_1 := u_1'][V_n]\neg[V_{n-1}]\neg(I_n \wedge u_1 = u_1') \end{array} \tag{2.7}$$

2.3 Modelo algorítmico

Esta seção apresenta as principais características do modelo algorítmico, as diferenças em relação ao modelo refinado e as suas obrigações de prova.

O modelo algorítmico é o modelo final da especificação no método B. Ele tem o papel de especificar o modelo somente com construções das linguagens de programação e tornar a especificação completamente determinística.

Diferentemente do modelo funcional ou refinado, o modelo algorítmico não permite o indeterminismo, pois isso é um requisito para possibilitar a geração de código de linguagem de programação. Esse nível, segundo [Schneider 2001], é restrito às seguintes construções:

- Atribuições simples na forma $[x := E]$;
- Composição de substituições em sequências: S ; T ;
- Condicional: **IF** E **THEN** S **ELSE** T **END** e a sua variação (**ELSIF**);
- Declaração de casos, usando a construção **CASE**;
- Laços[1]: **WHILE** E **DO** S **INVARIANT** I **VARIANT** V **END**;

[1] O nível de modelagem algorítmica é o único que permite o uso explícito da construção *while*. Portanto, os demais níveis de modelagem expressam laços de uma forma mais abstrata.

- Uso de variáveis locais: **VAR** x **IN** S **END**;
- Operações importadas dos modelos funcionais;
- Consulta de definições de outros modelos através da cláusula **SEES**.

Outro detalhe é que o modelo algorítmico sempre deve refinar um modelo funcional ou refinado, mas nunca pode ser posteriormente refinado. Além disso, esse modelo não possui variáveis abstratas próprias. Assim, para o contexto desse trabalho quem faz o papel dessas variáveis no modelo algorítmico são as variáveis abstratas dos modelos funcionais importados. Dessa forma no **INVARIANT** do modelo algorítmico é estabelecida a equivalência entre as variáveis do modelo refinado e as dos modelos funcionais importados. Todas as alterações sobre essas variáveis no modelo algorítmico são realizadas através de chamadas de operação.

As construções de obrigações de prova do modelo algorítmico são praticamente idênticas as do modelo refinado. Portanto, não se faz necessário apresentar novamente essas obrigações de prova.

2.3.1 Especificação de laços no método B

Nesta subseção é descrito como os laços são expressos com o método B, quais os pontos verificados e as garantias oferecidas através da verificação.

A construção **WHILE** (2.8) em B tem quatro partes: a de controle da iteração que é um teste condicional (P); a do corpo do laço (S) que é executado quando o teste é verdadeiro; a expressão (V), chamada variante do laço, que representa o número máximo de ciclos que podem ser executados e a fórmula (Iw), chamada invariante do laço, que deve ser válida cada vez que o teste condicional (P) é avaliado.

$$\boxed{\textbf{WHILE } P \textbf{ DO } S \textbf{ VARIANT } V \textbf{ INVARIANT } Iw \textbf{ END}} \tag{2.8}$$

A seguir é apresentado um exemplo prático de laço em B (2.9) que atribui zero a todos os valores do vetor "a", tendo "n" como o tamanho do vetor [Schneider 2001]. Note que o **INVARIANT** é verdadeiro para todas as iterações do laço e a cláusula **VARIANT** expressa um número de iterações que é estritamente decrescente. Essas duas cláusulas

são utilizadas para garantir a terminação e a pós-condição do laço. A construção dessas expressões é a parte mais difícil da especificação de laços.

$$
\begin{array}{l}
\textbf{WHILE}\, i < n\, \textbf{DO} \\
\quad i := i + 1; \\
\quad a.set\_v(i, 0) \\
\textbf{INVARIANT} \\
\quad \forall j.(j \in 1..i \Rightarrow a(j) = 0) \\
\quad \wedge\, i \leq n \wedge i \in \mathbb{Z} \\
\quad \wedge\, n \in \mathbb{Z} \\
\textbf{VARIANT}\, n - i\, \textbf{END}
\end{array}
\tag{2.9}
$$

Para certificar que o laço é corretamente especificado de acordo com o modelo, é necessário verificar se o laço satisfaz algumas regras. As regras que oferecem essa garantia de correção são as seguintes: o laço deve ser inicializado em um estado válido, ou seja, que satisfaz o invariante do laço; o fim da execução do laço deve satisfazer as pós-condições do modelo; o invariante do laço deve ser preservado pela execução do corpo do laço; e o variante deve ter um valor inteiro positivo e ser estritamente diminuído pela execução do corpo do laço. O conjunto dessas regras garante que o laço termina e satisfaz o invariante.

3 *Verificação em nível de assembly*

O atual capítulo apresenta a proposta de verificação formal de *software* até o nível de *assembly*. Entretanto, um pré-requisito para essa proposta é a modelagem das instruções *assembly* da plataforma de execução. Portanto, este capítulo também descreve a modelagem das instruções *assembly* de uma plataforma teórica simplificada [Medeiros 2007], chamada plataforma mínima, e apresenta um pequeno exemplo de *software* verificado até o nível de *assembly* da plataforma mínima. Esse capítulo portanto apresenta uma versão simplificada da abordagem proposta e constitui uma prova de conceito para o nosso projeto.

3.1 Plataforma mínima

A modelagem das instruções da plataforma mínima é baseada nas instruções dos atuais microcontroladores, os quais são computadores simplificados e baratos. Basicamente, microcontroladores são compostos por unidade de processamento, memória de acesso aleatório, memória somente de leitura e portas de entrada e saída. Muitas vezes possuem também temporizadores e outras funcionalidades avançadas facilitando o desenvolvimento de aplicações de tempo-real.

Para realizar a verificação em nível de *assembly* é necessário modelar o estado (ou seja, o conteúdo dos diversos elementos de memorização que possui) e as instruções (ou seja, o efeito de cada instrução sobre o estado). Assim, as instruções *assembly* e o estado da plataforma são representados respectivamente por operações e variáveis B.

O modelo apresentado nesta seção tem como objetivo apenas introduzir a ideia inicial da especificação do conjunto de instruções de uma plataforma. Então, algumas abstrações serão estabelecidas como: o tipo primitivo dos dados será representado pelo conjunto dos naturais e todas as instruções da plataforma estão definidas em um módulo de especificação chamado *Proc_Cont*. O capítulo 4 apresenta uma descrição detalhada de um

modelo de microcontrolador real.

A especificação dessa plataforma mínima agrupa em um único módulo a unidade de controle, lógica e aritmética, e a memória. A parte inicial da especificação é representada a seguir. A unidade de controle é a responsável pelo fluxo de execução das instruções. Para isso, a unidade contém dois registradores representados através das variáveis *pc* e *end*, as quais seus valores pertencem ao conjunto dos naturais. A variável *pc* (contador de programa) indica a instrução que o processador deve executar. O papel da variável *end* é declarar a posição do fim do programa. Assim, em um programa correto, o valor de *pc* nunca deve ser maior que o valor de *end* e isso é expresso no invariante do modelo. Em relação ao armazenamento dos dados, o modelo funcional contém a variável w, que simboliza um registrador de trabalho e a variável *memory_data*. A *memory_data* representa a memória de dados e é uma função injetora total, que leva dos naturais aos naturais, em que o elemento do domínio representa o endereço na memória e o seu elemento imagem correspondente representa o conteúdo da memória nesse endereço. Por exemplo: $memory\_data \leftarrow \{(0 \mapsto 5)\}$ ou $memory\_data\,(0) := 5$ significa que no endereço zero da memória foi atribuído o valor cinco e as demais posições permanecem inalteradas.

MACHINE

Proc_Cont

VARIABLES

memory_data, *w1*, *pc*, *end*

INVARIANT

$memory\_data \ : \ (\ \mathcal{N} \rightarrow \mathcal{N}\ ) \wedge$

$w1 \in \mathcal{N} \wedge$

$pc \in \mathcal{N} \wedge$

$end \in \mathcal{N} \wedge$

$pc \leq end$

Em geral, cada operação B representa uma instrução *assembly* da plataforma. A seguir serão apresentadas apenas as operações mais significativas da plataforma mínima.

A três operações seguintes são relacionadas com fluxo de execução das instruções. A primeira $init(pc\_, end\_)$ delimita o início e o fim do programa; com essa delimitação é possível verificar se o contador de programa está em um intervalo válido.

init($pc\_$,$end\_$)=
PRE $pc\_ \in \mathcal{N} \wedge end\_ \in \mathcal{N} \wedge pc\_ \leq end\_$
THEN $pc := pc\_ \;||\; end := end\_$
END;

O *goto*(*address*) é responsável por efetuar saltos no fluxo de execução das instruções e faz isso atribuindo um valor ao contador de programa.

goto($address$) =
PRE $address \in \mathcal{N} \wedge address \leq end$
THEN $pc := address$
END;

A *iszero*(*address*) verifica se o valor no endereço de memória recebido é zero ou não. Caso seja zero, o contador é incrementado uma vez, senão ele é incrementado duas vezes.

iszero($address$) =
PRE $address \in \mathcal{N} \wedge (pc + 2) \leq end$
THEN
  IF $memory\_data(address) = 0$
  THEN $pc := pc + 1$
  ELSE $pc := pc + 2$
  END
END;

A instrução *iszero* e outras em conjunto com o *goto* permitem que os programas tenham trechos de programa executados condicionalmente. Assim, comumente, as instruções condicionais vêm seguidas de duas instruções *goto*, determinando o que deve ser executado caso o resultado da avaliação seja verdadeiro ou falso. Por exemplo:

iszero(*1*)
goto(*address_true*)
goto(*address_false*)

A operação *move* copia o valor em um endereço (*address1*) específico da memória de dados para um outro endereço (*address2*).

move(*address1*,*address2*) =
PRE $address1 \in \mathcal{N} \wedge address2 \in \mathcal{N}$
$\wedge\ (pc + 1) \leq end$
THEN
$memory\_data(address2) := memory\_data(address1)$
|| $pc := pc + 1$
END;

A modelagem da plataforma mínima contém ainda outras operações de diferentes grupos: controle de fluxo das instruções, atribuição ou cópia de dados e aritméticas. Todas as operações do modelo funcional *Proc_Cont* estão no anexo deste trabalho.

O modelo de plataforma mínima apresentado foi completamente desenvolvido e provado através da ferramenta AtelierB, o que garante que nenhuma instrução viola a consistência do modelo. Obviamente, esse é um modelo teórico e bastante simplificado. Entretanto, essa simplificação é muito importante para facilitar o entendimento, principalmente sobre a especificação das instruções do microcontrolador Z80 apresentado no capítulo 4.

3.2 Verificação em nível de *assembly*

Quando engenheiros de *software* desejam entender a natureza de um sistema de computador que precisam construir, eles geralmente constroem um modelo desse sistema [Wordsworth 1996]. Com base nessa ideia, esta seção apresenta a proposta de [Dantas et al. 2008] e um exemplo simples, que foi verificado formalmente até o nível de *assembly*.

Antes de apresentar o exemplo, será fornecida uma visão geral dos passos de desenvolvimento formal B estendido até o nível de *assembly*. A seguir, tem-se a Figura 6 com os passos. Essa segue a mesma ideia daqueles explicados no capítulo 2. Contudo, para cada modelo algorítmico B (4) existe um outro modelo algorítmico (4.5) correspondente, denominado B *assembly* que refina o mesmo modelo que o modelo algorítmico B (4).

Esse modelo adicional (4.5) pode ser gerado automaticamente a partir do *assembly* resultante da compilação tradicional (6) ou do modelo algorítmico B *assembly* (4).

- A partir do *assembly* (6) não se faz necessária a construção de um compilador, apenas de uma ferramenta para fazer uma transformação simples de *assembly* binário (6) para *assembly* simbólico (4.5), isso porque a geração para uma linguagem de pro-

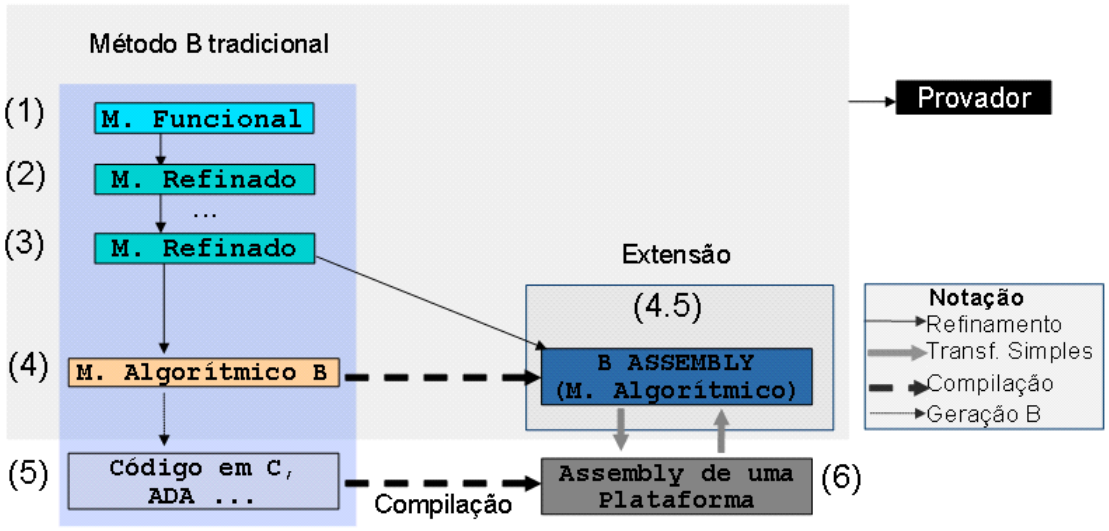

Figura 6: Etapas de verificação B estendida.

gramação (5) e a compilação para *assembly* (6) são suportadas pelas ferramentas atuais.

- Na outra forma, que é partindo do modelo algorítmico B (4) para o modelo algorítmico B *assembly* (4.5), é necessária a construção de um novo compilador e da ferramenta de transformação simples do modelo *assembly* B (4.5) para *assembly* (6). Aparentemente esta abordagem é mais interessante, pois o compilador participa mais ativamente do processo de verificação e pode adicionar ou construir fórmulas lógicas mais simples e específicas, utilizando as anotações (invariantes e variantes) do modelo algorítmico B para construir as anotações do modelo *assembly*.

Logo, de uma forma ou de outra é possível colocar-se em posição de verificar o refinamento do modelo algorítmico B *assembly* em relação ao seu modelo refinado, o que possibilita garantir a coerência semântica com base nas leis de refinamento.

A extensão do método B proposta é considerada mais confiável, porque aumenta o nível de verificação formal até o resultado da compilação. Na abordagem tradicional do método B, ao concluir a verificação no nível do modelo algorítmico B é possível, em ferramentas como o AtelierB, gerar código de linguagem de programação. Dessa forma, o processo de geração de código e compilação podem introduzir erros de difícil detecção. Na abordagem desenvolvida, o modelo B *assembly* é verificado formalmente e já está em um nível muito próximo ao da linguagem de máquina (código binário). Por mais que ainda seja necessário aplicar transformações, essas não envolvem risco, pois basicamente fazem

uma tradução simples, de um programa em linguagem de *assembly* com sintaxe concreta (B *assembly*) para outro de sintaxe concreta (*assembly*).

3.2.1 Especificação do modelo do programa *Swap*

O exemplo desta seção ainda é muito simples, porém é um bom exemplo para explicação da abordagem explorada neste trabalho. O objetivo do programa *swap* é realizar a troca de valor entre duas variáveis. Na Figura 7 é apresentado o modelo funcional do programa *Swap_Ini*. Um fato interessante é que nesse nível de abstração as operações são atômicas, ou seja, os valores de "a" e "b" são alterados simultaneamente, o que não existe em linguagens algorítmicas. Logo, esse modelo é refinado em outro, nesse caso o modelo algorítmico *Swap_Alg*.

MACHINE
Swap_Ini
VARIABLES
a0, *b0*
INVARIANT
$a0 \in \mathbf{INT} \wedge b0 \in \mathbf{INT}$
INITIALISATION
$a0, b0 := 0, 0$
OPERATIONS
swap =
$a0, b0 := b0, a0$
END

Figura 7: Modelo funcional do programa *swap*.

O modelo *Swap_Alg*, na Figura 8, estabelece o valor das variáveis importadas com as do modelo funcional e todas as manipulações são efetuadas sobre as variáveis importadas. Isso acontece porque a modelagem algorítmica do método B não permite o uso de variáveis abstratas próprias. Dessa forma, instanciações (importações) de um modelo funcional (*Var_Natural*) são utilizadas como variáveis. Esse modelo também é ilustrado na Figura 8 e possui uma variável (*value*) e operações de acesso (*set*) e (*get*). Assim, o modelo algorítmico pode possuir muitas variáveis através da importação de outros modelos.

No entanto, para garantir que o modelo algorítmico refina corretamente o modelo funcional, é indispensável a especificação da relação de igualdade entre as variáveis do modelo importado (*Var_Natural*) e o funcional (*Swap_Ini*), isso é descrito na cláusula **INVARIANT** do modelo *Swap_Alg*.

Na operação *swap* são criadas duas variáveis locais para realizar a troca de valores,

isso porque de fato não é possível implementar a operação atômica para troca de valores entre variáveis, então a especificação foi construída de uma forma sequencial. Portanto, torna-se evidente que o principal papel do nível algorítmico é restringir a especificação às construções implementáveis.

```
IMPLEMENTATION
  Swap_Alg
REFINES Swap_Ini
IMPORTS
  a0.Var_Natural, b0.Var_Natural
INVARIANT
  a0 = a0.value ∧ b0 = b0.value
INITIALISATION
  BEGIN
    a0.set(0);
    b0.set(0)
  END
OPERATIONS
  swap =
  BEGIN
    VAR av, bv
    IN
      av ← a0.get;
      bv ← b0.get;
      a0.set(bv);
      b0.set(av)
    END
  END
END
```

```
MACHINE Var_Natural
VARIABLES value
INVARIANT value ∈ 𝒩
INITIALISATION
   value := 0
OPERATIONS
  set (new_value) =
   PRE new_value ∈ 𝒩
   THEN value := new_value
   END;
  res ← get =
      res:= value
END
```

Figura 8: Modelo algorítmico do programa *Swap* e o modelo importado *Var_Natural*.

Com o modelo algorítmico B, então, é possível gerar automaticamente a especificação em *assembly* correspondente. Na Figura 9 é ilustrada a parte inicial da modelagem algorítmica em B *assembly*. As relações entre as variáveis do modelo mais abstrato com o valor dos endereços de memória são declaradas no invariante de estado. Estabelecendo essa relação, o ambiente de verificação do método B é capaz de garantir se a semântica do modelo B *assembly* (*Swap_Assembly*) é coerente com o modelo funcional (*Swap_Ini*). Além disso, o método B garante o fim da execução do modelo do *software*.

A operação *swap* em B *assembly* é representada na Figura 10. O controle do fluxo de instruções *assembly* é realizado através do **WHILE** e **CASE**, e a troca de valor entre as variáveis é executada por meio de três instruções *move*. Portanto, a modelagem em *assembly* é maior e mais complexa. Consequentemente, sua especificação é relativamente

IMPLEMENTATION *Swap_Assembly*
REFINES *Swap_Ini*
IMPORTS
micro.Proc_Cont
INVARIANT
$a0 = micro.memory\_data(0) \wedge b0 = micro.memory\_data(1)$
INITIALISATION
BEGIN
micro.init_data(0,0);
micro.init_data(1,0)
END

Figura 9: Cabeçalho do modelo B *assembly* do programa *swap*.

mais difícil, principalmente porque se faz necessário escrever de forma manual as cláusulas **INVARIANT** e **VARIANT** do laço (**WHILE**) . E essas cláusulas podem ser longas, como mostrado na Figura 10. O **INVARIANT** é um pouco mais difícil de escrever, porque ele deve restringir e satisfazer todos os possíveis estados das variáveis do **WHILE**, por isso tamanha complexidade na sua construção. Além disso, esse **INVARIANT** deve especificar a correspondência entre as variáveis do modelo abstrato e o mais concreto em cada iteração.

A Tabela 2 contém as estatísticas da verificação dos modelos.

Módulo	O. de P. óbvias	O. de P. não óbvias	Provadas
swap_ini	10	0	100%
swap_alg	11	0	100%
swap_assembly	35	63	100%
TOTAL	46	76	100%

Tabela 2: Tabela das estatísticas da obrigação de prova do exemplo *swap*.

Nesse exemplo, que é bem simples, foram verificadas 63 obrigações de prova não óbvias da modelagem B *assembly*, oito dessas foram realizadas no provador interativo, mas quatro entre as oito necessitaram de uma maior interação com provador. Apesar da dificuldade, novas técnicas de verificação automática foram disponibilizadas em [Clearsy 2009] e serão citadas nos capítulos seguintes.

O trabalho de [Medeiros 2007] contém pequenos exemplos de programas especificados e verificados até o nível de *assembly*. O conjunto das construções algorítmicas utilizadas nesses programas possui a mesma capacidade de computação que a máquina de Turing[1].

[1] A máquina de Turing é um dispositivo teórico, conhecido como máquina universal, que foi concebido pelo matemático britânico Alan Turing. A tese Turing é que essa máquina pode computar programas de qualquer computador digital.

```
swap =
VAR local_pc, local_end IN
  local_pc := 0;
  local_end := 3;
  micro.init(local_pc,local_end);
  WHILE local_pc < local_end DO
    BEGIN
      CASE local_pc OF
        EITHER 0 THEN micro.move(0,2)
        OR 1 THEN micro.move(1,0)
        OR 2 THEN micro.move(2,1)
        END
      END;
      local_pc ← micro.get_pc
    END
  INVARIANT
    local_pc ≥ 0 ∧ local_pc ≤ 3 ∧
    local_pc = micro.pc ∧
    local_end = micro.end ∧
    (local_pc = 0 ⇒
      (micro.memory_data(0) = a0 ∧ micro.memory_data(1) = b0)) ∧
    (local_pc = 1 ⇒
      (micro.memory_data(0) = a0 ∧ micro.memory_data(1) = b0
        ∧ micro.memory_data(2) = a0)) ∧
    (local_pc = 2 ⇒
      (micro.memory_data(0) = b0 ∧ micro.memory_data(1) = b0
        ∧ micro.memory_data(2) = a0)) ∧
    (local_pc = 3 ⇒
       (micro.memory_data(0) = b0 ∧ micro.memory_data(1) = a0))
  VARIANT (local_end - local_pc)  END
END
```

Figura 10: Especificação da operação *swap* em modelagem B *assembly*.

Como todos os programas podem ser computados nessa máquina teórica, então, esse é um indício da capacidade de modelar o *assembly* de qualquer programa correto.

4 *Modelagem de instruções assembly em B*

Como explicado anteriormente, este trabalho tem como objetivo contribuir com um dos grandes desafios da computação, que é automatização da verificação de código até o nível de *assembly*. Portanto, o objetivo final dessa linha de pesquisa é desenvolver um compilador capaz de produzir código verificado formalmente. No entanto, um pré-requisito deste projeto é modelar a plataforma de execução e amadurecer a metodologia de verificação citada no capítulo 3. Dessa forma, este capítulo apresenta a modelagem de uma plataforma usando o método B e cita algumas técnicas usadas no processo de prova das condições de verificação desse modelo.

A modelagem apresentada neste capítulo é de um microcontrolador real e baseada no modelo da plataforma mínima, citada na seção 3.2. Diferentemente do modelo da plataforma mínima, o modelo do microcontrolador contém vários detalhes de especificação e uma vasta quantidade de instruções. Logo, muitas decisões de projeto foram efetuadas com minucioso cuidado, considerando pequenos detalhes e os seus possíveis impactos no modelo, a fim de evitar futuras alterações em conceitos elementares. Adicionalmente, aproveitando-se dos mecanismos de divisão de responsabilidades do método B, foram construídas bibliotecas de módulos básicos.

As bibliotecas são de tipos de dados de *hardware* e de tipos de dados inteiros. Essas bibliotecas possuem definições sobre conceitos comuns nas plataformas de microcontroladores e microprocessadores, e são utilizadas nos modelos das plataformas 8051 e PIC ainda não concluídos. Consequentemente, as bibliotecas promovem o reuso da modelagem, o que acelera o desenvolvimento da pesquisa.

O atual capítulo está organizado da seguinte forma: as seções 4.1 e 4.2 contêm a modelagem dos componentes básicos de *hardware* e tipos de dados; a seção 4.3 descreve o modelo de microcontrolador desenvolvido. Enfim, a última seção contém as considerações finais.

4.1 Biblioteca de *hardware*

A biblioteca de *hardware* é utilizada para representar entidades elementares, tais como: *bits*, vetores de *bits* e os tipos de dados derivados. Essa biblioteca também provê funções lógicas e aritméticas sobre esses tipos. Dessa forma, a biblioteca de *hardware* e outras podem ser empregadas na construção de modelos de diversas plataformas.

A especificação desta biblioteca obteve a colaboração direta dos pesquisadores David Déharbe e Stephenson Galvão, pois eles iniciaram a modelagem das plataformas: PIC 16C432 e A8051. Isto incentivou a modularização e o reuso da biblioteca, o que forçou a criação de funções com baixo acoplamento e alta coesão[1]. Essas características também melhoram todo o processo de verificação.

A biblioteca de *hardware* modela cinco módulos diferentes, como mostrado na Figura 11. Os detalhes sobre cada um dos módulos da biblioteca serão apresentados nas próximas subseções. A princípio é importante conhecer o módulo *Power2* e a notação da relação *SEES*, onde o módulo *M1* possui acesso as definições do módulo *M2*. O módulo *Power2* contém definições que facilitam o cálculo da potenciação, uma operação que o provador do AtelierB possui dificuldade para trabalhar, usada para converter valores da representação binária para inteira no módulo *Bit_Vector_Arithmetics*.

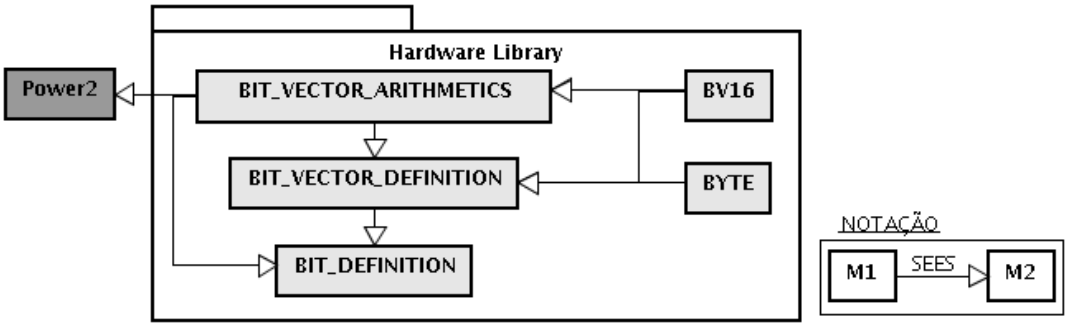

Figura 11: Dependência entre os módulos da biblioteca de *hardware*.

4.1.1 Definições para representar e manipular *bits*

As entidades definidas no módulo *BIT_DEFINITION* são: tipos para *bits*, operadores lógicos em *bits* (negação, conjunção, disjunção, disjunção exclusiva) e as funções de conversão entre *booleano* e *bits*. A apresentação dos diferentes módulos é realizada de forma

[1] Acoplamento é o grau em que os módulos são relacionados ou dependentes de outros módulos. Coesão é o grau em que cada módulo desempenha articuladamente uma função geral. Os módulos devem ter alta coesão e baixo acoplamento.

livre, e apenas serão mostradas as definições essenciais, sendo omitidas as construções de estruturação de B.

Primeiro, *bits* são modelados como um intervalo de inteiros: $BIT = 0..1$. A negação é uma função unária em *bits* e é definida como:

$$bit\_not \in BIT \rightarrow BIT \wedge \forall(bb).(bb \in BIT \Rightarrow bit\_not(bb) = 1 - bb)$$

O módulo também provê lemas da negação que são úteis para usuários da biblioteca no desenvolvimento de provas:

$$\forall(bb).(bb \in BIT \Rightarrow bit\_not(bit\_not(bb)) = bb)$$

Conjunção é uma função sobre bits e é definida como:

$$bit\_and \in BIT \times BIT \rightarrow BIT \wedge$$
$$\forall(b1, b2).(b1 \in BIT \wedge b2 \in BIT \Rightarrow$$
$$((bit\_and(b1, b2) = 1) \Leftrightarrow (b1 = 1) \wedge (b2 = 1)))$$

O módulo provê os seguintes lemas para conjunção, também:

$$bit\_and(0, 0) = 0;\ bit\_and(0, 1) = 0;$$
$$bit\_and(1, 0) = 0;\ bit\_and(1, 1) = 1;$$
$$\forall(b1, b2).(b1 \in BIT \wedge b2 \in BIT \Rightarrow$$
$$(bit\_and(b1, b2) = bit\_and(b2, b1))) \wedge$$
$$\forall(b1, b2, b3).(b1 \in BIT \wedge b2 \in BIT \wedge b3 \in BIT \Rightarrow$$
$$(bit\_and(b1, bit\_and(b2, b3)) = bit\_and(bit\_and(b1, b2), b3)))$$
$$\forall(b1).(b1 \in BIT \Rightarrow (bit\_and(b1, 1) = b1));$$
$$\forall(b1).(b1 \in BIT \Rightarrow (bit\_and(b1, 0) = 0));$$

Este módulo possui definições de *bit_or* (disjunção) e *bit_xor* (ou exclusivo) assim como lemas para seus operadores. Estes são padronizados e suas expressões em B são similares às de *bit_and*, então elas são omitidas aqui, mas elas estão disponíveis completamente para consulta no repositório do projeto (`http://code.google.com/p/b2asm`).

Finalmente, a conversão de booleanos para *bits* é simplesmente definida como:

$$bool\_to\_bit \in \mathbf{BOOL} \rightarrow BIT \wedge bool\_to\_bit = \{\mathbf{TRUE} \mapsto 1, \mathbf{FALSE} \mapsto 0\}$$

É importante enfatizar que todos os lemas desse módulo foram automaticamente verificados pelo provador de teoremas incluído no AtelierB [Clearsy 2009]. Ou seja, nenhuma das provas necessitou de interação do usuário.

4.1.2 Representação e manipulação de vetor de *bits*

Sequências são predefinidas em B como funções em que o domínio é um intervalo inteiro iniciando em 1 (um). Os índices de vetores de *bits* geralmente iniciam em 0 (zero) e o modelo proposto obedece a esta convenção. O modelo aplica um deslocamento onde necessário e esse deslocamento é importante para usar as funções já predefinidas de sequências do método B. Deste modo, nós definimos vetores de *bits* como sequências não vazias de *bits* e *BIT_VECTOR* é o conjunto de todas essas sequências, que é definido abaixo como:

$$BIT\_VECTOR = \mathbf{seq_1}(BIT)$$

A função *bv_size* retorna o tamanho dado de um vetor de *bits*. Esta é simplesmente um encapsulamento para a função *size* predefinida das sequências. Associada à função *bv_size* existem lemas que declaram o tamanho do vetor de *bits* de acordo com o tipo de dado, o que evita o provador de teoremas utilizar a definição da função *size* e contar número de elementos do conjunto facilitando o processo de prova.

$$bv\_size \in BIT\_VECTOR \rightarrow \mathcal{N}_1 \wedge$$
$$bv\_size = \lambda\, bv.(bv \in BIT\_VECTOR \mid \mathbf{size}(bv))$$

Nós também definimos duas funções *bv_set* e *bv_clear* que, dado um vetor de *bits* e uma posição do vetor de *bits*, retorna o vetor de bits recebido com a atribuição de 1 ou 0 na posição correspondente. Para obter o valor de um *bit* a partir de um vetor de bits, a função *bv_get* foi definida. Como essas funções são bem semelhantes, então, abaixo apresentamos apenas a definição da primeira.

$$bv\_set \in BIT\_VECTOR \times \mathcal{N} \rightarrow BIT\_VECTOR \wedge bv\_set =$$
$$\lambda\, v, n.(v \in BIT\_VECTOR \wedge n \in \mathcal{N} \wedge n < bv\_size(v) \mid v \leftarrow\!\!+ \{n+1 \mapsto 1\})$$

Adicionalmente, o módulo provê definições para combinações da lógica clássica dos vetores de *bits*: *bv_not*, *bv_and*, *bv_or* and *bv_xor*. Contudo, somente as duas primeiras são apresentadas abaixo. Observe que o domínio dos operadores binários é restrito para pares de vetores de bits de mesmo tamanho, o que evita eventuais erros na especificação.

$$bv\_not \in BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$$
$$bv\_not = \lambda\, v.(v \in BIT\_VECTOR \mid \quad \lambda\, i.(1..bv\_size(v)) \mid bit\_not(v(i))) \wedge$$
$$bv\_and \in BIT\_VECTOR \times BIT\_VECTOR \rightarrow BIT\_VECTOR \wedge$$
$$bv\_and = \lambda\, v_1, v_2.(v_1 \in BIT\_VECTOR \wedge v_2 \in BIT\_VECTOR \wedge$$
$$\quad bv\_size(v_1) = bv\_size(v_2) \mid \lambda\, i.(1..bv\_size(v_1)) \mid bit\_and(v_1(i), v_2(i)))$$

Vários outros lemas foram criados sobre operações de vetores de *bits*. Estes lemas

expressam propriedades sobre o tamanho do resultado das operações e as propriedades de associatividade e comutatividade; eis alguns deles:

$$\forall v.(v \in BIT\_VECTOR \Rightarrow bv\_size(bv\_not(v)) = bv\_size(v))$$
$$\forall v.(v \in BIT\_VECTOR \Rightarrow bv\_not(bv\_not(v)) = v)$$
$$\forall v_1, v_2.(\{v_1, v_2\} \subseteq BIT\_VECTOR \wedge bv\_size(v_1) = bv\_size(v_2) \Rightarrow bv\_size(bv\_and(v_1, v_2)) = bv\_size(v_1))$$
$$\forall v_1, v_2.(\{v_1, v_2\} \subseteq BIT\_VECTOR \wedge bv\_size(v_1) = bv\_size(v_2) \Rightarrow bv\_size(bv\_and(v_1, v_2)) = bv\_size(v_2))$$
$$\forall v_1, v_2.(\{v_1, v_2\} \subseteq BIT\_VECTOR \wedge bv\_size(v_1) = bv\_size(v_2) \Rightarrow bv\_size(bv\_and(v_1, v_2)) = bv\_and(v_2, v_1))$$
$$\forall v_1, v_2, v_3.(\{v_1, v_2, v_3\} \subseteq BIT\_VECTOR \wedge bv\_size(v_1) = bv\_size(v_2) \wedge bv\_size(v_1) = bv\_size(v_3) \Rightarrow bv\_and(bv\_and(v_1, v_2), v_3) = bv\_and(v_1, bv\_and(v_2, v_3)))$$

4.1.3 Modelando *bytes* e vetores de *bits* de tamanho 16

Vetores de *bits* de tamanho 8 são *bytes* e este conceito é representado na máquina *BYTE_DEFINITION*. Esta forma uma entidade comum em projeto de *hardware*. Portanto, nós elaboramos as seguintes definições:

$$BYTE\_WIDTH = 8 \wedge BYTE\_INDEX = 1 \,..\, \text{BYTE\_WIDTH} \wedge$$
$$PHYS\_BYTE\_INDEX = 0 \,..\, (BYTE\_WIDTH\text{-}1) \wedge$$
$$BYTE = \{\ bt \mid bt \in BIT\_VECTOR \wedge bv\_size(bt) = BYTE\_WIDTH\} \wedge$$
$$\times \{0\}$$

O *BYTE_INDEX* é o domínio do *byte* modelado. Este inicia em 1 para obedecer a definição de sequências do método B. Entretanto, as funções de *byte* encapsulam o acesso e suas funções usam o *PHYS_BYTE_INDEX*. O *BYTE* é um tipo especializado de *BIT_VECTOR*, mas esse contém um limite de tamanho. Portanto, algumas definições mais específicas foram criadas no módulo *BYTE_DEFINITION* para complementar a modelagem. Similarmente, o tipo *BV16* foi criado para o vetor de *bits* de tamanho 16.

4.1.4 Aritmética de vetores de *bits*

Os vetores de *bits* são usados para representar e combinar números inteiros com e sem sinal. Nossa biblioteca define um módulo (*BIT_VECTOR_ARITHMETICS*) com definições e funções para manipular estes dados, por exemplo, a função *bv_to_nat* mapeia

vetores de *bits* para números inteiros naturais.

$$bv\_to\_nat \in BIT\_VECTOR \rightarrow \mathcal{N} \wedge$$
$$bv\_to\_nat = \lambda\, v.(v \in BIT\_VECTOR \mid \sum i.(i \in \mathsf{dom}(v).v(i) \times 2^{i-1}))$$

Este módulo ainda possui definições de funções inversas e lemas associados, veja um exemplo:

$$\forall\, n.(n \in \mathcal{N} \Rightarrow bv\_to\_nat(nat\_to\_bv(n)) = n)$$

Enfim, como esta é uma biblioteca de baixo nível, então a maioria dos conceitos de mais alto nível são estabelecidos com base nesta biblioteca. Ela também disponibiliza um conjunto de lemas, os quais foram definidos e provados com o objetivo de reduzir o tamanho das provas dos modelos que usam as definições providas.

4.2 Biblioteca dos tipos de dados inteiros

O conjunto de instruções do microcontrolador tem tipos de dados comuns. Estes tipos são localizados na nossa biblioteca de tipos. Cada módulo tem uma função para manipular e converter seus dados. Existem seis tipos de dados comuns representados pelos módulos, veja na tabela 3.

Nome	*UCHAR*	*SCHAR*	*USHORTINT*	*SSHORTINT*	*BYTE*	*BV16*
Faixa	0..255	-128..127	0..65.535	-32.768..32.767	–	–
Tamanho	1 byte	1 byte	2 bytes	2 bytes	1 bytes	2 bytes

Tabela 3: Descrição dos tipos de dados inteiros

Geralmente, os módulos de mais alto nível necessitam apenas detalhar ou redefinir os conceitos pré-definidos. O tipo *BYTE* é um subconjunto do *BIT_VECTOR*, dessa forma suas definições podem ser complementadas. Por exemplo, a função *bv_to_nat* do módulo *BIT_VECTOR_ARITHMETICS* é especializada para *byte_uchar*, então esta função pode ser definida como segue:

$$byte\_uchar \in BYTE \rightarrow UCHAR \wedge$$
$$byte\_uchar = \lambda(v).(v \in BYTE | bv\_to\_nat(v))$$

Uma forma alternativa, mais específica para vetores de *bits* de tamanho fixo e simples da função *byte_uchar* é apresentada abaixo. Essa forma evita o uso do somatório, portanto pode simplificar o processo de verificação.

$byte\_uchar \in BYTE \rightarrow UCHAR \wedge$
$byte\_uchar = \lambda(v0).(v0 \in BYTE | 2^7 * bv\_get(v0,7) + 2^6 * bv\_get(v0,6)$
$+2^5 * bv\_get(v0,5) + 2^4 * bv\_get(v0,4) + 2^3 * bv\_get(v0,3)$
$+2^2 * bv\_get(v0,2) + 2 * bv\_get(v0,1) + bv\_get(v0,0))$

A função inversa é facilmente definida como *uchar_byte*.

$uchar\_byte \in UCHAR \rightarrow BYTE \wedge$
$uchar\_byte = (byte\_uchar)^{-1}$

Nós também criamos os lemas abaixo. Várias outras funções e lemas semelhantes foram criadas para os outros tipos de dados.

$\forall(val).(val \in UCHAR | byte\_uchar(uchar\_byte(val)) = val) \wedge$
$\forall(by).(by \in BYTE | uchar\_byte(byte\_uchar(by)) = by)$

Por fim, a definição da nossa biblioteca de tipos reusa definições da biblioteca de *hardware*. Isto melhora a confiança e agiliza o processo de prova. Pois, quando vários projetistas utilizam uma determinada biblioteca, esta tende a melhorar suas definições e corrigir eventuais erros. Além disso, no contexto da especificação formal elas tendem a fornecer lemas mais úteis e padronizados, o que reduz sensivelmente o tempo do processo de verificação. Finalmente, essas bibliotecas são genéricas o suficiente para permitir especificar diferentes plataformas de *hardware*.

4.3 Modelagem do microcontrolador Z80

O modelo do microcontrolador desenvolvido em B não tem como objetivo modelar e verificar todas as suas características. Em resumo, o objetivo da modelagem é representar o conjunto de instruções *assembly* da plataforma. Em outras palavras, a modelagem do microcontrolador pretende especificar o efeito das instruções e eventos externos sobre o estado da memória e registradores, o que é suficiente para a verificação em nível de *assembly*. Outras questões intrínsecas do hardware foram ignoradas na modelagem, por exemplo, tempo, voltagem e frequência. O objetivo principal da modelagem não é verificar se o *hardware* é corretamente projetado. Por outro lado, o modelo é capaz de verificar várias características. Entretanto existem abordagens mais eficientes para verificar propriedades sobre os componentes de *hardware*, tais como *pipeline* [Velev 2004], protocolos de barramento [Ramasamy, Cukier e Sanders 2003], circuitos digitais [Aljer et al. 2003] entre outras.

O Z80 é um importante microcontrolador CISC criado em 1976 pela *Zilog* [Zilog 2001].

Nesta época, ele obteve uma grande aceitação do mercado, porque ele suportava todas as 72 instruções do *8080*, um popular microcontrolador da *Intel Corporation* no momento. Então, o conjunto de 158 instruções do Z80 lembra um pouco o conjunto da arquitetura x86. Além disso, o Z80 possui uma vasta documentação e funcionalidades interessantes, o que faz ele ainda ser utilizado atualmente. Por outro lado, existem pesquisadores modelando outros conjuntos de instruções com abordagens de trabalho semelhante.

Este trabalho focaliza os esforços na modelagem do Z80 e adota o método B por possuir funcionalidades interessantes. Nosso trabalho restringiu-se a um modelo simples para obter resultados nos prazos estabelecidos, e ao mesmo tempo permitiu vislumbrar a aplicação de B para modelar os atuais microprocessadores. O custo de modelagem e verificação ainda é alto, porém este trabalho cita algumas funcionalidades e técnicas usadas com o método B para diminuir esse custo.

A capacidade de modularização é uma das funcionalidades do método B que criou um bom mecanismo de separação das responsabilidades dos módulos. A Figura 12 contém um diagrama com os relacionamentos de dependência do projeto da plataforma Z80 com as demais bibliotecas desenvolvidas. A fim de não poluir o diagrama, as dependências entre os módulos de projetos diferentes são representadas por uma única seta entre os projetos.

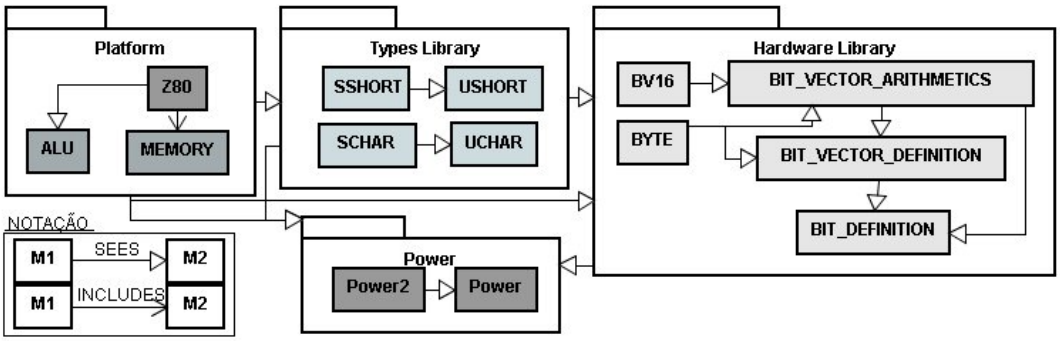

Figura 12: Diagrama de dependência entre os módulos em uma visão geral.

O módulo principal, apresentado a seguir e chamado Z80, inclui uma instância do módulo de memória e acessa as definições dos módulos: *Power2*, ALU e bibliotecas básicas. Mais detalhes sobre o módulo ALU serão apresentados mais a frente.

MACHINE

Z80

INCLUDES

MEMORY

SEES

BIT_DEFINITION, BIT_VECTOR_DEFINITION,
BYTE_DEFINITION, BV16_DEFINITION,
UCHAR_DEFINITION, SCHAR_DEFINITION,
SSHORT_DEFINITION ,USHORT_DEFINITION, POWER2, ALU

4.3.1 Modelando registradores e portas de entrada e saída

Os registradores internos do Z80 contêm centenas *bits* de leitura e escrita de memória. Isto inclui dois conjuntos de seis registradores que podem ser usados individualmente como registradores de 8 *bits* e em pares de 16 *bits*. O registrador de trabalho é representado pela variável *rgs8*. O domínio de *rgs8* é um conjunto *id_reg_8* formado por identificadores de registradores de 8 *bits*. Esses registradores podem ser acessados em pares, formando 16 *bits*, resultando em outro conjunto de identificadores chamado *id_reg16*. Esses conjuntos são declarados na cláusula **SETS** que pode ser vista a seguir. Também é importante conhecer o principal registrador do Z80, o acumulador (*rgs8*(*a0*)) que é usado para operações aritméticas, lógicas, de entrada e saída e leitura/escrita.

SETS

id_reg_8 = { *a0* , *f0* , *f_0* , *a_0* ,
b0 , *c0* , *b_0* , *c_0* ,
d0 , *e0* , *d_0* , *e_0* ,
h0 , *l0* , *h_0* , *l_0* } ;
id_reg_16 = { *BC* , *DE* , *HL* , *SP* , *AF* }

A CPU Z80 inclui um conjunto alternativo de registros de memória: acumulador, *flag* e de uso geral. A lista a seguir contém uma descrição de outros elementos que possuem estados:

sp - Registrador apontador da pilha

pc - Registrador contador de programa

ix e *iy* - Registradores de indexação

*i_* - Registrador de interrupção

*r_* - Registrador de atualização

iff1, *iff2* - Registradores usados para controlar as interrupções

im - Um par de *bits* para definir o modo de interrupção

i_o_ports - Portas de entrada e saída

Esses elementos são variáveis de estado e a tipagem deles é especificada através do invariante. Note que o tipo dessas variáveis está na representação binária para oferecer uma proximidade semântica com modelo real, exceto a variável *pc* que representa o contador de programa, pois o *pc* é atualizado constantemente e uso da representação dos números inteiro facilita o processo de verificação evitando constantes conversões de dados.

INVARIANT

$rgs8 \in id\_reg\_8 \rightarrow BYTE \wedge$
$pc \in USHORT \wedge sp \in BV16 \wedge ix \in BV16 \wedge iy \in BV16 \wedge$
$i\_ \in BYTE \wedge r\_ \in BYTE \wedge$
$iff1 \in BIT \wedge iff2 \in BIT \wedge$
$im : (BIT \times BIT) \wedge$
$i\_o\_ports \in BYTE \rightarrow BYTE$

4.3.2 Registrador de *flag*

Outro importante elemento é o registrador "z" ($rgs8(f0)$), que é usado como um registrador de flag. Esse registrador usa somente seis *bits* para representar o estado geral do resultado de cada instrução. Cada *bit* é atribuído um (1) quando seu significado for verdadeiro para o contexto da última instrução executada, caso contrário, zero (0). De acordo com o manual oficial [Zilog 2001], os *bits* 3 e 5 não são utilizados e os outros têm o seguinte significado:

$\boldsymbol{bv\_get(rgs8(f0), 0)}$ - O bit de *carry*;

$\boldsymbol{bv\_get(rgs8(f0), 1)}$ - O bit de subtração(1)/soma(0) indica qual operação aritmética ocorreu;

$\boldsymbol{bv\_get(rgs8(f0), 2)}$ - O bit de paridade par ou *overflow* tem seu significado dependente do tipo de instrução;

bv_get(rgs8(f0), 4) - O bit *half carry* indica se ocorreu *carry* entre os *bits* 3 e 4;

bv_get(rgs8(f0), 6) - O bit zero indica se o resultado foi zero;

bv_get(rgs8(f0), 7) - O bit de sinal indica se o resultado foi negativo.

Os valores desses *bits* são bastantes úteis ao programador. Adicionalmente, eles permitem impor propriedades de segurança no modelo.

Propriedade 4.3.1 (Garantindo a Ausência de *Overflow*) *Para garantir que não aconteça overflow, o desenvolvedor pode adicionar esta expressão* $(bv\_get(rgs8(f0), 0) \neq 1 \wedge bv\_get(rgs8(f0), 2) \neq 1)$ *na cláusula INVARIANT. Por padrão, o overflow pode acontecer sem problemas, todavia, muitas vezes isto pode ser perigoso, pois um overflow indica que o resultado de uma operação de um determinado tipo de dado está fora do intervalo válido do seu tipo de dado original, ou seja, o resultado obtido é truncado. Dessa forma, o desenvolvedor pode prevenir seu acontecimento. Entretanto, esta restrição certamente torna mais difícil o processo de verificação, pois ela implica que as pré-condições das operações devem ser mais fortes.*

4.3.3 Funções de manipulação de dados

Existem algumas funções auxiliares específicas da plataforma para a definição semântica das instruções. A função *bv_ireg_plus_d* é usada para calcular o endereçamento no modo indexado. Esta recebe o valor do registrador (*ix* ou *iy*) e o deslocamento para retornar a soma; o resultado é um endereço de memória deslocado, veja a definição abaixo.

$$bv\_ireg\_plus\_d : (BV16 \times SCHAR \rightarrow BV16) \wedge$$
$$bv\_ireg\_plus\_d = \lambda\ (\ ix\_iy\ ,\ disloc\ )\ .\ (\ ix\_iy \in BV16 \wedge disloc \in SCHAR \mid$$
$$ushort\_bv16\ (\ (bv16\_ushort\ (\ ix\_iy\ ) + disloc\ )\ \mathbf{mod}\ 2^{16})\ )$$

Uma outra função derivada é *bv_9ireg_plus_d0*, esta retorna o valor do endereço de memória retornado pela função *bv_ireg_plus_d* e sua definição é similar.

Existe uma função específica para atualizar o registrador *flag* (*rgs8(f0)*), essa é chamada *update_reg_flag*. Ela é tipada da seguinte forma: $update\_flag\_reg \in (BIT \times BIT \times BIT \times BIT \times BIT \times BIT) \rightarrow (\{f0\} \times BYTE)$. A sua definição segue abaixo:

$$update\_flag\_reg = \lambda\ (s7\ ,\ z6\ ,h4\ ,pv2\ ,n1\ ,c0\ ).$$
$$(\ s7 \in BIT \wedge z6 \in BIT \wedge h4 \in BIT \wedge pv2 \in BIT \wedge n1 \in BIT \wedge c0 \in BIT$$
$$|(\ f0 \mapsto [c0,\ n1,\ pv2,\ 1,\ h4,\ 1,\ z6,\ s7])\ )$$

4.3.4 Memória de programas, pilha e dados

O Z80 possui uma única memória para armazenar instruções de programa, dados da pilha e dados de trabalho. A memória é especificada no módulo *Memory*. Ela tem endereçamento de 16 *bits* e cada endereço armazena um *byte*. Por conseguinte, a memória é muito simples, mas um cuidado adicional deve ser tomado para preservar a sua consistência: os dados devem ser armazenados na sua região de memória específica. O invariante abaixo contém a tipagem da variável que representa a memória:

INVARIANT

$mem \in BV16 \rightarrow BYTE$

Por padrão, as instruções do Z80 podem acessar todos os endereços de memória, mas isso pode ser arriscado. Para adicionar mais segurança, é importante que as instruções tenham acesso a regiões de memória limitado. Portanto, o projetista pode especificar regiões de endereços para restringir o acesso as instruções. As regiões podem ser especificadas como mostrado abaixo. Por outro lado, o modelo real especificado é fiel as características do Z80, logo esta restrição está desabilitada no modelo real.

$$PROGRAM\_R\_ADDRESS = 0..16384 \land DATA\_R\_ADDRESS = 16385..49151 \land STACK\_R\_ADDRESS = 49152..65535$$

Propriedade 4.3.2 (Garantindo a ausência de sobreposição das regiões) *Para garantir que as regiões de memória são exclusivas, o projetista pode opcionalmente adicionar a seguinte expressão*[2] *no invariante:*

$$PROGRAM\_R\_ADDRESS \cap DATA\_R\_ADDRESS \cap STACK\_R\_ADDRESS = \{\}$$

Propriedade 4.3.3 (Preservando a consistência da memória) *Em geral, o acesso a algumas regiões de memória é perigoso. Por este motivo, em cada instrução existe uma pré-condição que verifica se o endereço de memória, que será atualizada, pertence a sua região de memória correta. Por exemplo, a instrução de programa PUSH permite escrever somente na região da pilha (STACK_R_ADDRESS). Logo, um programa correto não permite a instrução PUSH escrever fora da região STACK_R_ADDRESS.*

Para manipular os valores da memória existem operações que encapsulam o acesso. As operações permitem alterar o valor de um único endereço da região de memória de trabalho (*updateAddressMem*) ou um conjunto (*updateMem*). Também existem operações específicas para manipular a região da pilha.

[2]Essa expressão implica em pré-condições mais fortes nas modelagem das instruções, portanto essa expressão aparece como comentário na modelagem atual.

4.3.5 Unidade lógica e aritmética

Existem muitas funções definidas no módulo *ALU*. Em geral, essas funções são construídas através da composição de outras funções mais básicas. Por exemplo, a função *half8UCHAR* é usada para obter o *half carry* do valor. Assim, a função *half8UCHAR* é usada na função *add8UCHAR*.

$$half8UCHAR \in UCHAR \rightarrow UCHAR \wedge$$
$$half8UCHAR = \lambda\ (ww).(ww \in UCHAR \mid ww \ \mathbf{mod}\ 2^4)$$

A função *add8UCHAR* faz a soma entre elementos do tipo *UCHAR*. Ela recebe um *bit* de *carry* e dois valores *UCHAR* e retorna respectivamente a soma binária, o *bit* do sinal, o bit *carry*, o *bit half carry* e o *bit* que indica se o resultado é zero (0).

$$add8UCHAR \in (BIT \times UCHAR \times UCHAR) \rightarrow$$
$$(UCHAR \times BIT \times BIT \times BIT \times BIT\ ) \wedge$$
$$add8UCHAR = \lambda\ (carry, w1,\ w2).$$
$$(carry \in BIT \wedge w1 \in UCHAR \wedge w2 \in UCHAR \mid$$
$$(((carry + w1 + w2\ )\ \mathbf{mod}\ 2^8\ ),$$
$$bool\_bit(\ carry + uchar\_schar(w1) + uchar\_schar\ (w2) < 0),$$
$$bool\_bit(\ carry + w1 + w2 > UCHAR\_MAX),$$
$$bool\_bit(\ carry + half8UCHAR(w1) + half8UCHAR(\ w2) \geq 2^4),$$
$$bool\_bit(\ (\ (carry + w1 + w2\ )\ \mathbf{mod}\ 2^8\ ) = 0))\ \ )$$

Uma função similar para a operação de subtração é *substract8UCHAR*. Existem as mesmas funções para o tipo *SCHAR*, eles são respectivamente *add8SCHAR* e *substract8SCHAR*, todas essas funções são de 8 *bits* (*BYTE*), mas as funções aritméticas para 16 *bits* (*BV16*) também são definidas similarmente.

Outras funções mais simples são tipadas e explicadas abaixo:

- $inc \in BYTE \rightarrow BYTE$ - Recebe um *byte* e retorna seu incremento. Existe uma função similar chamada *dec*;
- $instruction\_next \in USHORT \rightarrow USHORT$ - Recebe o valor do *pc* atual e retorna seu incremento;
- $is\_negative \in BYTE \rightarrow BIT$ - Retorna 1 se o *byte* recebido é negativo, caso contrário retorna zero (0);
- *update_refresh_reg* - Recebe um *byte* e retorna seu incremento, porém considera o valor com apenas sete *bits*.

Todas as constantes e funções definidas nos módulos *BYTE* e *BV16* são visíveis no módulo *ALU*. Então, o módulo *ALU* se limita a definir funções e constantes mais específicas da plataforma.

4.3.6 Modelando as operações de mudança de estado

As instruções do Z80 são classificadas nos grupos: carga e troca de dados; transferência e busca de blocos de dados; lógica e aritmética de dados; rotação e deslocamento de dados; manipulação de *bits* (*set, reset, test*); chamada de função (*jump, call* e *return*); entrada e saída de dados; e controle da fluxo do *cpu*.

O principal módulo (*Z80*) tem dois tipos de operações: um tipo representa as ações externas e o outro as instruções *assembly* do microcontrolador. As ações externas são apresentadas na subseção 4.3.6.1. Um exemplo simples de instrução é a $LD\_(nn)\_A$[3] como mostrado abaixo[4]. Ela simplesmente copia o valor do registrador de trabalho ($rgs8(a0)$) para um novo endereço (nn) fornecido como parâmetro. Perceba que a instrução tem como pré-condição a verificação do tipo do parâmetro e se ele pertence a sua região de memória correta. Ela usa a operação *updateAddressMem* do módulo de memória que recebe um endereço de memória e seu novo valor de memória. A instrução ainda incrementa o contador de programa (pc) e atualiza o registrador de atualização ($r\_$).

LD_9nn0_A (nn) =
  PRE $nn \in USHORT \quad \wedge \quad nn \quad \in DATA\_R\_ADDRESS$
  THEN
  updateAddressMem ($ushort\_bv16$ (nn) , $rgs8$ ($a0$)) ||
  $pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END

As demais instruções têm uma estrutura similar. As mudanças mais significativas da especificação das instruções variam de acordo com o modo de endereçamento, o grupo da instrução e seus subgrupos. Essas pequenas variações geram uma quantidade grande de instruções e a maioria delas é omitida neste documento. No entanto, todas as instruções do Z80 foram modeladas.

[3] O AtelierB não permite usar parênteses nos seus identificadores, então os caracteres "(" e ")" são substituídos respectivamente por "9" e "0" na especificação real.

[4] Por padrão, todos os parâmetros das operações do módulo principal são ou elementos pré-definidos no modelo ou valores inteiros na representação decimal.

4.3.6.1 Modelando as ações externas

As ações externas são eventos que podem ocorrer independente do contexto interno da plataforma. Essas ações mudam o estado do microcontrolador, por exemplo, a atualização das portas de entrada e saída e as requisições de interrupções. As ações externas também são modeladas por operações e os seus nomes são prefixados com "*ext_*" e seguido do nome da ação. Existem apenas quatro ações: *ext_update_io_ports*, *ext_NMI*, *ext_INT* e *ext_Reset*. A *ext_update_io_ports* apenas atualiza o estado de uma porta de entrada e saída, veja:

ext_update_io_ports(*address*,*value*)=
PRE *address* $\in$ *UCHAR* $\wedge$ *value* $\in$ *SCHAR* **THEN**
io_ports (*uchar_byte* (*address*)) := *schar_byte* (*value*)
END

As demais ações externas são relacionadas com as interrupções. As interrupções permitem que um dispositivo externo suspenda uma rotina da *CPU* e, em seguida, a *CPU* inicia uma outra rotina de serviço. Essa rotina de serviço pode trocar dados ou sinais entre a *CPU* e dispositivos externos. Quando a rotina é finalizada, então a *CPU* volta para o ponto da rotina que foi interrompida e continua sua execução.

Para as interrupções, os seguintes elementos são importantes: os *flip-flops* de interrupção (*iff1* e *iff2*), os tipos de interrupção (mascarável e não-mascarável), o modo de interrupção (atribuído através das instruções: *IM0*, *IM1*, *IM2*) e o registrador de interrupção (*i_*).

O par de *bits iff1* e *iff2* controla as interrupções mascaráveis (*INT*). Quando o *iff1* é um, a interrupção é habilitada, caso contrário esta é desabilitada. O *iff2* é usado somente como um local de armazenamento temporário para *iff1*. O *iff1* é valorado com um ou zero através das instruções *EI* e *DI*.

A essência do efeito dos dois tipos de interrupção é apresentada a seguir. Contudo, antes é importante conhecer algumas definições: *sp* representa o apontador da pilha, *sp_minus_two* é igual à *dec_BV16*(*dec_BV16*(*sp*)), *sp_minus_one* é igual à *dec_BV16*(*sp*), *pc_high* é a sequência dos 8 *bits* mais significativos do contador de programa e *pc_low* é a sequência dos menos significativos.

NMI - Interrupção não-mascarável - Essas interrupções não podem ser desabilitadas pelo programador. Então, quando um dispositivo faz uma requisição desse tipo, o *sp* é empilhado, o *pc* recebe $66H$ (102 em decimal), *iff1* é zerado, *iff2* armazena *iff1* e o

registrador de atualização é incrementado. A especificação dessa interrupção contém as substituições apresentadas a seguir.

updateStack({ $(sp\_minus\_two \mapsto pc\_low)$,$(sp\_minus\_one \mapsto pc\_high\ )$ })||
$sp := sp\_minus\_two$ || $pc := 102$ || $iff1{:=}0$ || $iff2{:=}\ iff1$ ||
$r\_ := update\_refresh\_reg(r\_)$

INT - Interrupção Mascarável - Ela é geralmente reservada para funções importantes que podem ser habilitadas e desabilitadas pelo programador. Quando acontece uma interrupção mascarável, ambos *iff1* e *iff2* recebem zero, o *sp* é empilhado, o registrador de atualização é incrementado e os outros efeitos dependem do modo de interrupção.

- O modo 0 é compatível com o 8080 e o valor de *im* deve ser igual à ($0 \mapsto 0$). Quando uma interrupção não-mascarável acontece nesse modo, o contexto atual é empilhado e uma instrução de um *byte* provida de um dispositivo externo é executada, geralmente uma instrução RST. O código de instrução é recebido através do barramento de dados e ele é representado por um parâmetro inteiro chamado *byte_bus*.

- O modo 1 é o mais simples e o valor de *im* deve ser igual à ($0 \mapsto 1$). Simplesmente, quando uma interrupção não-mascarável acontece, o contexto atual é empilhado e o contador de programa recebe $38H$ (56 em decimal).

- O modo 2 é o mais poderoso com relação ao fluxo de execução das instruções e o valor de *im* deve ser igual à ($1 \mapsto 1$). Quando uma interrupção não-mascarável acontece, o contexto atual é empilhado e uma chamada indireta pode ser feita para algum endereço de memória. Isso acontece da seguinte forma: o contador de programa recebe na parte mais significativa o registrador $i\_$ e na parte menos significativa o *byte_bus* com o último *bit* zerado.

A parte essencial da operação *ext_INT*[5] é representada a seguir:

[5] O *byte_bus* é um parâmetro da operação *ext_INT*, que representa as interrupções mascaráveis, é mostrada a seguir.

IF $im = (\ 0 \mapsto 0\ )$ **THEN**
IF $byte\_bus \in opcodes\_RST\_instruction$
THEN
$pc := byte\_bus$ - 199 ||
updateStack({ $stack(\ sp\_minus\_one) \mapsto pc\_low$,
$stack(sp\_minus\_two) \mapsto pc\_high$ }) ||
$sp := sp\_minus\_two$ || $r\_ := update\_refresh\_reg(r\_)$
ELSIF $byte\_bus = opcode\_\ldots\_instruction$
...
END
ELSIF $im = (\ 0 \mapsto 1\ )$ **THEN**
updateStack({ $stack(sp\_minus\_one) \mapsto pc\_low$,
$stack(sp\_minus\_two) \mapsto pc\_high$ }) ||
$sp := sp\_minus\_two$ || $r\_ := update\_refresh\_reg(r\_)$||
$pc := 56$
ELSIF $im = (\ 1 \mapsto 1\ )$ **THEN**
$pc := bv16\_ushort(byte\_bv16(\ i\_\ ,bv\_clear(rotateleft(uchar\_byte(byte\_bus)),0)))$||
updateStack({ $stack(\ sp\_minus\_one) \mapsto pc\_low$,
$stack(sp\_minus\_two) \mapsto pc\_high$ }) ||
$sp := sp\_minus\_two$ || $r\_ := update\_refresh\_reg(r\_)$
END

RESET - Apenas zera os registradores de trabalho. Ela é muito usada para reinicializar o contexto de execução. As suas substituições são apresentadas a seguir:

iff1 :=0 || *iff2*:=0 || *im*:= $(0 \mapsto 0)$ ||
pc:=0 || $i\_$:= [0,0,0,0,0,0,0,0] ||
rgs8 := $rgs8 \leftarrow \{\ (a0 \mapsto [0,0,0,0,0,0,0,0]\ ,\ (f0 \mapsto [0,0,0,0,0,0,0,0]\ \}$ ||
$r\_$:= [0,0,0,0,0,0,0,0] || *sp* := [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

4.3.6.2 Modelando instruções de controle das portas de entrada e saída

O Z80 tem um conjunto extenso de instruções de entrada e saída e 256 portas para dispositivos. Ele pode transferir blocos de dados entre os dispositivos de E/S e alguns dos registradores internos ou endereços de memória.

As instruções básicas são *IN* para entrada e *OUT* para a saída de dados. Nessas instruções, é necessário fornecer o endereço do dispositivo e o registrador ou endereço de

memória. Por exemplo, a instrução $IN\ A,(n)$ coloca os dados do dispositivo da porta n no registrador acumulador ($rgs8(a0)$). O parâmetro n deve ser um número pertencente a $UCHAR$ e, normalmente, ele é fornecido pelo registrador c ($rgs8(c0)$), como na instrução $IN\ rr,(C)$ que é representada pela operação abaixo. A pré-condição dessa instrução garante que o elemento rr passado é pertencente ao conjunto id_reg_8 e diferente do registrador de *flag* ($f0$). Ela recebe um identificador de registro rr; nesse registro é armazenado o valor do endereço da porta indicado pelo registrador c. Além disso, incrementa o contador de programa e o registrador de atualização e ajusta o registrador de *flag* de acordo com o valor recebido através das funções $is\_negative, is\_zero, parity\_even, update\_flag\_reg$.

PRE $rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**

ANY

$negative$, $zero$, $half\_carry$, pv , add_sub , $carry$

WHERE

$negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge pv \in BIT \wedge$

$add\_sub \in BIT \wedge carry \in BIT \wedge$

$negative = is\_negative\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge$

$zero = is\_zero\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge$

$half\_carry = 0 \wedge$

$pv = parity\_even\ (\ io\_ports\ (\ rgs8\ (\ c0\ )\ )\ ) \wedge$

$add\_sub = \ 0 \wedge$

$carry = z\_c$

THEN

$rgs8 := rgs8 \leftarrow \{\ (\ rr \mapsto io\_ports\ (\ rgs8\ (\ c0\ )\ )\ )\ ,$

$update\_flag\_reg(negative,zero,half\_carry,pv,add\_sub,carry)\}||$

$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$

END

As principais instruções de saída funcionam de forma análoga, por exemplo: $OUT(n), A$ ou $OUT(C), r$. Existem várias outras instruções de E/S, por outro lado essas são as mais comumente utilizadas.

4.4 Regras no processo de prova

O provador de teoremas do AtelierB não consegue automaticamente realizar todas as provas mais complexas. Nesses casos é necessário escrever regras para ajudar na construção da prova. Algumas das regras abaixo parecem triviais e podem ser facilmente ser

deduzidas. No entanto, elas realizam um passo de prova maior que o normal, o que ajuda a concluir a prova eficientemente.

A lógica usada no processo de prova é, por natureza, incompleta. Isso quer dizer que não existe um sistema de regras que possa ser usado para derivar todos os teoremas. Portanto, é necessário aplicar incrementos ao sistema de regras. No entanto, existe um risco associado ao uso de regras, pois é possível utilizar regras inconsistentes. Nesse sentido, algumas das regras usadas foram verificadas e outras avaliadas criteriosamente.

A regra é uma fórmula com a seguinte forma: $A \Rightarrow B$. "A" é chamado o antecedente da regra, "B" é chamado o consequente da regra. Uma fórmula f diz-se casar com uma fórmula g, se a partir de g é possível obter f substituindo todas as variáveis curingas[6] com os valores das fórmulas.
Por exemplo, a expressão g casa com a fórmula f:
Expressão g: $aa + (bb/ee - (cc + dd) * aa)$
Fórmula f: $x + (y - z * x)$

A fórmula abaixo diz que a soma de palavras de 16 *bits* sem sinal de um valor "c" com "d" é diferente de "c" com "e", desde que pertençam ao mesmo tipo de valores e sejam diferentes. Esta fórmula foi verificada através do provador interativo.

$$d < 65536 \wedge e < 65536 \wedge d >= 0 \wedge e >= 0 \wedge f \in BIT \wedge not(d = e)$$
$$\Rightarrow not(ushort\_bv16(add16USHORT(f, c, d)) =$$
$$ushort\_bv16(add16USHORT(f, c, e)))$$

A fórmula abaixo diz que um valor "e" somado com um valor não neutro é diferente de "e". Esta fórmula não foi verificada utilizando o provador interativo, pois envolve definições complexas para o provador, contudo o leitor pode avaliar essa fórmula consultando as definições das funções.

$$c \in BIT \wedge e \in BV16 \wedge f \in USHORT \wedge$$
$$not(c + f = 0) \wedge not(c + f = 65536)$$
$$\Rightarrow not(ushort\_bv16(add16USHORT(c, bv16\_ushort(e), f)) = e)$$

A fórmula seguinte pode ser derivada da fórmula anterior, ela apenas muda o tipo do elemento "e" e introduz uma função de conversão. Esta fórmula não foi verificada utilizando o provador interativo, contudo o leitor pode avaliá-la, de modo análogo a fórmula anterior.

[6]Uma variável curinga pode assumir qualquer valor (literal, expressão, etc.) e é denotada através de um único caractere do alfabeto latino.

$$c \in BIT \wedge e \in USHORT \wedge f \in USHORT \wedge$$
$$not(c + f = 0) \wedge not(c + f = 65536)$$
$$\Rightarrow not(ushort\_bv16(add16USHORT(c, e, f)) = ushort\_bv16(e))$$

A fórmula seguinte é útil para verificar se uma sequência de *bits* é um *byte*. Esta fórmula foi verificada através do provador interativo.

$$a \in BIT \wedge b \in BIT \wedge c \in BIT \wedge d \in BIT \wedge$$
$$e \in BIT \wedge f \in BIT \wedge g \in BIT \wedge h \in BIT$$
$$\Rightarrow [a, b, c, d, e, f, g, h] \in BYTE$$

A fórmula seguinte é similar a anterior, contudo para uma sequência de *bits* de tamanho 16. Esta fórmula não foi verificada através do provador interativo. Contudo, a verificação da fórmula anterior é um forte indicativo da correção desta, pois a única diferença entre as duas é o tamanho da sequência.

$$a \in BIT \wedge b \in BIT \wedge c \in BIT \wedge d \in BIT \wedge$$
$$e \in BIT \wedge f \in BIT \wedge g \in BIT \wedge h \in BIT \wedge$$
$$i \in BIT \wedge j \in BIT \wedge l \in BIT \wedge m \in BIT \wedge$$
$$n \in BIT \wedge o \in BIT \wedge p \in BIT \wedge q \in BIT$$
$$\Rightarrow [a, b, c, d, e, f, g, h, i, j, l, m, n, o, p, q] \in BV16$$

A fórmula abaixo define a tipagem da função *update_flag_reg*. O leitor pode analisar a fórmula consultando a definição dessa função na seção 4.3.3. Esta fórmula também foi verificada através do provador interativo.

$$a \in BIT \wedge b \in BIT \wedge c \in BIT \wedge$$
$$d \in BIT \wedge e \in BIT \wedge f \in BIT$$
$$\Rightarrow update\_flag\_reg(a, b, c, d, e, f) \in (\{f0\} \times BYTE)$$

A fórmula seguinte é útil para verificar as substituições de três elementos na variável *rgs8*, em que *rgs8* é representado através da variável curinga k. Essa fórmula tem no antecendente a tipagem e a distinção entre os elementos do domínio do novo conjunto. Os elementos de distinção são necessários para não violar a definição de função. O número de elementos de distinções cresce de acordo com o número de elementos do novo conjunto. Dessa forma, para a sobrescrita de 7 novos elementos são necessários 21 elementos de distinções, o que dificulta o trabalho do provador automático. Então, essa mesma fórmula é reescrita em mais 4 versões diferentes para ser aplicável à sobrescrita até sete elementos novos. A fórmula foi verificada através do provador interativo na versão com a sobrescrita de 2 novos elementos.

$$b \in id\_reg\_8 \wedge d \in BYTE \wedge f \in id\_reg\_8 \wedge$$
$$g \in BYTE \wedge h \in id\_reg\_8 \wedge i \in BYTE \wedge$$
$$not(b = f) \wedge not(b = h) \wedge not(f = h) \wedge k : (id\_reg\_8 \rightarrow BYTE)$$
$$\Rightarrow k \mathbin{<\!\!+} \{b \mapsto d, f \mapsto g, h \mapsto i\} \in id\_reg\_8 \mathbin{+\!\!\!\!\rightarrow} BYTE$$

Essas regras facilitam o processo de prova, no entanto um cuidado especial deve ser tomado, pois a inserção de uma regra incorreta pode introduzir erros. Entretanto, algumas vezes, o provador apresenta deficiências e o mais indicado é utilizar provadores diferentes e estratégias de prova diferentes. Várias estratégias foram utilizadas através do provador interativo[7], contudo elas não foram suficientes para resolver todas obrigações de prova pendentes. Além disso, o autor desse trabalho não tinha conhecimento de todos os recursos oferecidos pelo AtelierB, o que resultou em um tempo maior de desenvolvimento do modelo.

Várias das provas realizadas interativamente foram custosas, pois o provador às vezes demonstrava dificuldades mesmo em obrigações de provas triviais. Nessas situações muitas horas de trabalho foram consumidas, porque o usuário era forçado a realizar um grande número de passos básicos para concluir a prova. Isso é normal acontecer, porque nenhum provador de teoremas é eficiente para todos os casos.

Portanto, o uso de diferentes provadores pode minimizar esse empecilho. Nesse objetivo, existem trabalhos iniciados para o método B suportar diferentes provadores [Clearsy 2009, Marinho et al. 2008], o que certamente pode minimizar o uso de regras.

4.5 Considerações finais sobre o modelo do Z80

Esta seção faz uma síntese sobre todo o processo de modelagem da plataforma. Para construir este modelo formal foi necessário inicialmente estudar todo o conjunto de instruções do Z80 e as características relacionadas com manipulação dos dados. Para remover eventuais dúvidas do manual oficial [Zilog 2001] foi utilizado o seu principal simulador [Soso 2002]. A atividade de codificar o modelo em B não foi difícil, apenas necessitou conhecer bem a plataforma. Entretanto, o processo de verificação do modelo foi um trabalho exaustivo, porque o modelo envolve muitos detalhes de baixo nível e o número de operações é grande. Porém, o modelo formal da plataforma ofereceu uma série de benefícios e proporcionou uma boa experiência no método B, o que certamente agiliza bastante a especificação e verificação de novas plataformas. A verificação formal do modelo traz

[7]Provador interativo é uma ferramenta com uma interface gráfica que recebe o auxílio do usuário no processo de prova.

uma série de garantias, pois as obrigações de provas geradas pelas ferramentas asseguram a verificação: dos tipos de dados, das propriedades de segurança e se as expressões utilizadas são bem definidas (EBD)[8]. Adicionalmente, o projetista tem uma grande flexibilidade para criar novas e específicas propriedades de segurança, o que é muito útil para ajustar a verificação de acordo com a necessidade exigida. Para obter todos esses benefícios foi necessário realizar uma quantidade expressiva de provas.

A Tabela 4 mostra a distribuição do número de obrigações de provas não óbvias e EBD do projeto Z80 separado em grupos. Um dos grupos que contém mais provas é o de manipulação de *bits*. Apesar da grande quantidade de provas, esse grupo foi rapidamente provado, pois ele utiliza vários lemas providos pela biblioteca de *hardware*, o que acontece também com o grupo de lógica e aritmética e a biblioteca de tipos.

Group	**Provas Não Obvias**	**Provas de EBD**
Entrada e Saída	54	296
Lógica e Aritmética	134	413
Manipulação de Bits	160	742
Ações Externas	51	84
Gerais	140	615
Inicialização, Propriedades, Asserções	68	169
	Subtotal 607	**Subtotal** 2319

Tabela 4: Estatísticas das provas organizadas em grupos

O nível de automatização do processo de verificação foi considerado muito satisfatório, pois a metodologia e suas ferramentas foram bem exploradas; caso a verificação obtivesse uma baixa automatização, o processo poderia ter sido significativamente lento e difícil, porque algumas técnicas e conceitos que oferecem bons resultados requerem conhecimentos relativamente novos e avançados. Naturalmente, as estatísticas confirmam o bom grau de automação, pois aproximadamente 89% das obrigações de prova não óbvias e 99% das EBD foram verificadas automaticamente.

Uma série de técnicas e fatores influenciam no processo de verificação. Muitas vezes, o simples cuidado na modelagem facilita o processo, por exemplo a racionalização do uso do operador *overwrite* ($\lhd\!\!+$) reduziu sensivelmente o número de obrigações de provas. A modularização do projeto ajudou a construir duas simples bibliotecas, que definem importantes lemas básicos, e também separou as instruções em módulos diferentes, o que colaborou para a verificação em paralelo. Para realizar as provas foram utilizados quatro

[8]Um expressão é chamada bem definida (ou não ambígua) quando atribuí uma única interpretação ou valor.

computadores, cada um com dois *cores*, e isto aumentou em muitas vezes a capacidade de processamento.

A fim de reduzir ainda mais o custo de verificação, comandos de prova[9] foram utilizados. Um comando de prova possui uma fórmula, uma espécie de padrão, e uma sequência de passos de prova. A fórmula serve para selecionar as obrigações de provas que podem ser resolvidas pela sequência de passos de prova. Esses comandos são muito úteis principalmente quando existem obrigações de provas semelhantes, pois as ferramentas são capazes de reusar os comandos para diferentes obrigações de prova. Portanto, poucos comandos de prova podem ser utilizados em muitas provas. Um bom exemplo é o conjunto de 17 comandos que rapidamente ajudou a verificar automaticamente 99% (2295) das obrigações de prova EBD; quando os comandos não eram utilizados, o provador verificou automaticamente apenas 50% dessas. Enfim, as técnicas mencionadas contribuíram para a verificação de 2926 obrigações de prova.

O processo de prova é a etapa que mais consumiu tempo, aproximadamente dois meses. Esse processo é complexo, porque é necessário construir, corrigir e provar vários predicados das bibliotecas de *hardware* e tipos. Por outro lado, quando as bibliotecas básicas já estão prontas, a modelagem e verificação de novas plataformas torna-se mais rápida e fácil.

A modelagem do conjunto de instruções microcontrolador é útil para documentar, simular e ser utilizada como referência para implementação do *hardware*, pois a modelagem é representada de forma bem definida, diferentemente de outras representações, por exemplo, descrições textuais. Essa modelagem pode ser vista como uma documentação que especifica as instruções *assembly* e operações externas que modificam o estado do microcontrolador, o que é muito útil para programadores *assembly*. A modelagem das instruções permitiu identificar alguns pequenos erros na documentação do Z80. A maioria dos erros identificados eram relacionados com as funções que atualizam o registrador de *flag*, pois algumas vezes o autor do manual copiava a descrição de uma instrução, porém ele esquecia de atualizar algum detalhe sobre a ação do registrador de *flag*, isso era identificado no processo de especificação e confirmado através da consulta de outras fontes [Soso 2002, Young 2003].

A modelagem das instruções, teoricamente, pode ser animada, ou seja, simulada. A modelagem do Z80 particularmente contém definições complexas ainda não adequada-

[9] Os comandos de provas são passos que apenas indicam um caminho para o provador, logo esses comandos não são capazes de introduzir hipóteses falsas.

mente suportadas em uma das mais estáveis ferramentas B de animação [Leuschel e Butler 2003]. Contudo, a modelagem da plataforma mínima apresentada na seção 3.1 é suportada, o que possibilita o usuário avaliar o comportamento de cada instrução e até mesmo uma sequência de instruções.

Este capítulo apresentou um exemplo de modelagem formal em B do conjunto de instruções *assembly* do Z80. Este exemplo fornece resultados interessantes, pois ele também ajudou a encontrar alguns erros e ambiguidades no manual oficial [Zilog 2001]. Dessa forma, o modelo da plataforma pode substituir ou melhorar a documentação utilizada por programadores *assembly*, porque o método B tem uma interessante e fácil notação. Além disso, o modelo da plataforma possibilita a verificação de propriedades de segurança, restringe-se ao uso de definições com tipagem correta e expressões bem definidas. Finalmente, o modelo da plataforma também é útil para especificação de sistemas até o nível de linguagem *assembly*.

5 *Estudo de caso*

Este capítulo apresenta um estudo de caso que realiza uma avaliação experimental da abordagem de computação verificada no arcabouço do método B [Dantas et al. 2008] com um projeto piloto. O objeto do projeto piloto é parte do sistema de teste de produção convencional em tanque. O principal objetivo desse estudo é a construção da modelagem funcional e o desenvolvimento por refinamento até o nível de *assembly* para a plataforma Z80.

A seção 1 apresenta uma introdução sobre o sistema de teste de produção em tanque e a seção seguinte descreve a modelagem B e B *assembly*. A seção 3 mostra a simulação do *software* desenvolvido, o que foi realizado para testar o seu funcionamento. Finalmente, a última seção apresenta as considerações finais sobre a modelagem desse estudo de caso.

5.1 Teste de produção em tanque

O teste de produção é "o processo usado para o acompanhamento do desenvolvimento da produção de um campo de petróleo, o qual é executado em cada poço deste campo numa frequência previamente definida" [Silva 2008]. Nesse processo são analisadas as características do óleo, gás e água extraídos, tais como: volume, concentração de água, temperatura, entre outras.

Esse tipo de teste possui um papel importante no processo de extração. Ele é fundamental para avaliar a capacidade de extração dos poços e para designar o pagamento dos impostos governamentais, o pagamento de *royalties* aos municípios, as participações especiais e as participações aos proprietários de terra. No entanto, atualmente, segundo registros dos testes no sistema de informação, 10% deles são falhos [Silva 2008]. Logo, é importante que esses testes obtenham a melhor precisão possível, a fim de garantir uma melhor qualidade, credibilidade e transparência.

Atualmente, importantes técnicas da engenharia de automação industrial são uti-

lizadas para oferecer uma melhor confiança do teste de produção. No entanto, técnicas de especificação formais ainda não são utilizadas, logo este trabalho tem como objetivo utilizá-las para prover garantias matemáticas sobre o correto funcionamento do sistema, principalmente com relação às partes mais críticas.

O sistema de teste de produção é utilizado para controlar e avaliar a produção de petróleo. Ele possui um sistema supervisório que calcula a produção de petróleo e controla as válvulas de acordo com os estados do sistema, do sensor de interface e do radar. O radar informa o nível total do fluido no tanque e o sensor de interface informa a concentração de água na emulsão em um ponto local independente da densidade, viscosidade e temperatura de operação, o que ajuda a identificação do nível da interface. Esse sistema é complexo e pode ser considerado sob vários pontos de vista, no entanto, ele é simplificado nessa proposta até o nível de abstração adequado para facilitar o entendimento. Uma série de técnicas e cuidados físicos deve ser aplicada para realização correta dos testes de produção. Dessa forma, o sistema de teste envolve uma considerável complexidade. Além disso, os trabalhos de [Silva 2008, Lima 2000] definem vários detalhes físicos e parâmetros para melhorar a qualidade dos testes, os quais foram obtidos e aprimorados após experimentos em campo. Esses detalhes podem ser consultados nos trabalhos citados, e nesse trabalho é apresentada apenas uma visão geral do problema e a especificação de uma parte do sistema.

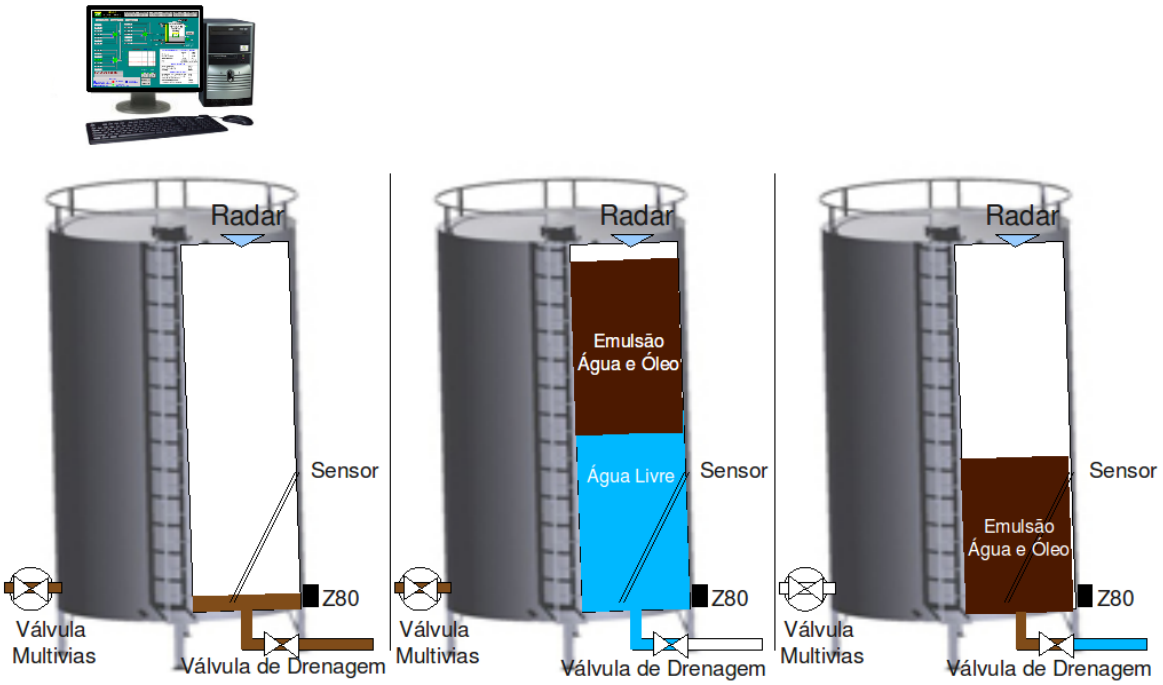

Figura 13: Fases do teste de produção.

O teste de produção, resumidamente, possui três fases de acordo com [Silva 2008].

Primeiramente, inicia a fase de condicionamento, o sistema de controle do tanque é atualizado com dados coletados do último teste, o sistema mantém o poço alinhado[1] e ao final dessa fase a válvula de drenagem é fechada. Em seguida, inicia a fase de enchimento e decantação, logo após o enchimento e a espera, que pode variar de 8 a 20 horas de decantação, começa a fase de drenagem. Nessa última fase, o sistema deve controlar a válvula de drenagem para não permitir fluir o óleo extraído e calcular a sua quantidade. E esse cálculo é o objeto de interesse do presente estudo de caso.

As três fases citadas são exibidas na mesma ordem na Figura 13. Nessa figura é possível observar os dispositivos utilizados no teste de produção e os estados do tanque em cada fase.

Na fase de drenagem, o nível de óleo desce relativamente rápido até aproximar-se do sensor de interface. Nesse instante a válvula de drenagem começa a fechar lentamente e, consequentemente, diminui a velocidade de saída do fluido. Todo esse cuidado é importante para evitar a formação de vórtice[2], o que afeta diretamente as propriedades do óleo extraído.

Para melhor entender a fase de drenagem é importante conhecer o conceito de BSW, a abreviação de *Basic Sediments and Water*, que representa a proporção de água e sedimentos no fluído extraído. De acordo com o valor do BSW histórico do poço e outras propriedades do fluído, o sistema estima o nível da interface água e óleo. Em seguida, o sistema passa a controlar a válvula de drenagem e a monitorar o estado do sensor e do radar. O sistema vai ajustando lentamente o nível de abertura da válvula de drenagem até o sensor de interface identificar a emulsão água e óleo. Finalmente, o processo de teste termina.

Como uma medição exata reduz sensivelmente a margem de erro, a técnica desenvolvida por [Silva 2008] pretende identificar com boa exatidão o atual nível da lâmina de óleo através das informações seguintes: valor do sensor de interface, nível do tanque obtido pelo radar e outros parâmetros obtidos através de experimentos. Porém, uma garantia matemática sobre o correto funcionamento do sistema é fundamental. Assim, um cuidado especial deve ser realizado nesse sistema.

Para essa finalidade, um *software* foi especificado, verificado, implementado e simulado. A princípio, o *software* desenvolvido pode atuar como um sistema redundante na plataforma Z80 para interagir com os dispositivos e o sistema supervisório. Esse *software*

[1]Um poço está alinhado quando as válvulas da tubulação direcionam o fluxo do poço para o tanque.
[2]Vórtice é disposição concêntrica e raiada do fluido, ou seja, redemoinho ou pequenas ondas.

apenas realiza um cálculo relativo a produção de óleo, com a finalidade de avaliar se o sistema está funcionando corretamente.

Aparentemente, o projeto de desenvolvimento desse *software* é uma atividade relativamente simples, porém como a técnica de verificação utilizada até o nível de *assembly* é completamente inovadora, então surgiram algumas dificuldades iniciais que levaram tempo para ser solucionadas. A seção seguinte apresenta mais detalhes sobre a modelagem desse *software*.

5.2 Modelagem B

A presente seção apresenta a modelagem B e o *software* do cálculo do fator de proporção de óleo bruto (emulsão água e óleo) e água livre produzidos.

Os fatores de proporção de óleo bruto e água livre são determinados através de informações obtidas na fase de drenagem. As informações utilizadas são os níveis do tanque: inicial (tanque cheio) e final (tanque com apenas óleo bruto) da fase de drenagem. Portanto, a subtração desse dois valores determina um fator de proporção de água livre produzida e o nível final do tanque determina um fator de proporção de óleo bruto produzido. Para determinar exatamente a quantidade de óleo e água livre, é necessário que o resultado seja multiplicado por um fator de correção, o qual deve representar as distorções e o formato do tanque, já que o tanque não tem um formato perfeitamente cilíndrico. Dessa forma, esse *software* embarcado torna-se genérico para qualquer formato de tanque.

A modelagem é iniciada com o modelo funcional *TestCalc*. Esse modelo contém duas variáveis *oil_factor* e *free_water_factor* que representam respectivamente o fator final de óleo e o de água livre. A seguir, é apresentado o invariante e a operação do modelo funcional. O invariante declara o tipo das variáveis como *UCHAR*, um inteiro positivo de 8 *bits*, ou seja, pertencente ao intervalo de 0 até 255. A operação recebe como parâmetro o valor inicial e final do nível do tanque e então calcula o fator de óleo e água livre. Uma pré-condição dessa operação é que o nível inicial do tanque cheio deve ser maior ou igual ao nível após a remoção da água livre. Isso é uma simplificação e evita que seja realizado tratamento de exceção até o nível de *assembly* para esse código.

INVARIANT

$oil\_factor \in UCHAR \wedge free\_water\_factor \in UCHAR$

OPERATIONS

update_factor(*initial_level*, *final_level*) =

PRE

$initial\_level \in UCHAR \wedge final\_level \in UCHAR \wedge$

$final\_level \leq initial\_level$

THEN

$free\_water\_factor := initial\_level$ - $final\_level$ ||

$oil\_factor := final\_level$

END

A operação *update_factor* do modelo funcional *TestCalc* é similar à sua modelagem algorítmica, exceto o fato das substituições não acontecerem em paralelo, mas acontecem em sequência no modelo algorítmico, então o refinamento da modelagem algorítmica foi verificado e a sua apresentação é omitida aqui.

Parte da modelagem B no nível de *assembly* é representada a seguir. Ela é especificada no modelo *TestCalc_basm* e possui a mesma semântica de manipulação das variáveis que o modelo abstrato (*TestCalc*), entretanto utiliza operações que representam instruções *assembly* de uma instância do modelo do Z80 para manipular a sua memória.

A cláusula invariante estabelece a relação das variáveis do modelo inicial (*free_water_factor* e *oil_factor*) com os valores dos endereços 2 e 3 da porta de entrada e saída; para estabelecer essa relação são utilizadas funções que convertem valores da representação binária para representação de inteiro e vice-versa. Dessa forma, a verificação do refinamento garante que as operações do modelo B *assembly* são semânticamente equivalentes às operações do modelo mais abstrato de acordo com a relação estabelecida entre as variáveis no invariante.

IMPORTS

Z80

INVARIANT

$byte\_uchar(io\_ports(uchar\_byte(2))) = (free\_water\_factor\ ) \wedge$

$byte\_uchar(io\_ports(uchar\_byte(3))) = (oil\_factor)$

A seguir é apresentada a operação *update_factor* utilizando instruções em nível de *assembly*. Essa operação possui a mesma assinatura que o modelo mais abstrato, e os parâmetros recebidos são convertidos para representação em binário e passados para a operação de atualização das portas do microcontrolador (*ext_update_io_ports*). Sucintamente, a sequência de instruções ilustradas a seguir realiza os seguintes procedimentos. As cinco primeiras instruções representam apenas a cópia dos dados externos ao microcontrolador para os registradores de memória "A" e "C". A instrução seguinte realiza uma subtração, então as demais copiam os fatores de proporção de água livre e óleo bruto respectivamente para as portas 2 e 3. O leitor pode consultar o anexo desse trabalho para entender detalhes da especificação de cada instrução e o invariante completo da operação *update_factor* ilustrada a seguir.

update_factor(*initial_level*, *final_level*) =
  ASSERT
    initial_level ∈ *UCHAR* ∧ *final_level* ∈ *UCHAR* ∧ *final_level* ≤ *initial_level*
  THEN
    VAR *local_pc* **IN** *local_pc* := 0; **set_pc**(*local_pc*);
      WHILE *local_pc* < 9 **DO**
        CASE *local_pc* **OF**
          EITHER 0 **THEN** **ext_update_io_ports**(0,*uchar_schar*(*initial_level*));
                IN_A_9n0(0)
          OR 1 **THEN** **LD_r_r_**(*b0*,*a0*)
          OR 2 **THEN** **ext_update_io_ports**(1,*uchar_schar*(*final_level*));
                IN_A_9n0(1)
          OR 3 **THEN** **LD_r_r_**(*c0*,*a0*)
          OR 4 **THEN** **LD_r_r_**(*a0*,*b0*)
          OR 5 **THEN** **SUB_A_r**(*c0*)
          OR 6 **THEN** **OUT_9n0_A**(2)
          OR 7 **THEN** **LD_r_r_**(*a0*,*c0*)
          OR 8 **THEN** **OUT_9n0_A**(3)
          END **END**; *local_pc* ← **get_pc**
        INVARIANT
          local_pc ∈ 0 .. 9 ∧ *rgs8* ∈ *id_reg_8* → *BYTE*
          ∧ *r_* ∈ *BYTE* ∧ *io_ports* ∈ *BYTE* → *BYTE*
          ∧ *pc* ∈ 0 .. 9 ∧ *free_water_factor* ∈ *UCHAR* ∧
          (*local_pc* = 0 ⇒ (*pc* = 0 ∧ *instruction_next*(*pc*) = 1 ∧
              byte_uchar(*io_ports*(*uchar_byte*(2))) = *free_water_factor* ∧
              byte_uchar(*io_ports*(*uchar_byte*(3))) = *oil_factor*) ∧
              ...
          (*local_pc* = 9 ⇒ (*pc* = 9 ∧
              byte_uchar(*io_ports*(*uchar_byte*(0))) = *initial_level* ∧
              byte_uchar(*io_ports*(*uchar_byte*(1))) = *final_level* ∧
              byte_uchar(*rgs8*(*a0*)) = (*final_level*) ∧
              byte_uchar(*rgs8*(*c0*)) = (*final_level*) ∧
              byte_uchar(*rgs8*(*b0*)) = (*initial_level*) ∧
              byte_uchar(*io_ports*(*uchar_byte*(2))) = ((*initial_level* - *final_level*) **mod** 256) ∧
              byte_uchar(*io_ports*(*uchar_byte*(3))) = *final_level*))
        VARIANT (9 - *local_pc*) **END**
        END **END**
      END

O invariante do **WHILE** deve formalizar o estado e o mapeamento das variáveis do modelo abstrato com os valores dos endereços de memória relacionado. Portanto, para cada iteração do *while*, que é associada com o valor do contador de programa (*pc*), deve existir uma expressão para realizar essa formalização. A cláusula variante deve expressar o limite superior do número de instruções a ser executado para cada valor possível do contador de programa.

Um detalhe interessante desse modelo é que as instruções do Z80 podem receber valores inteiros, porém o modelo do Z80 representa internamente os dados na notação binária. Essa diferença na representação poderia dificultar bastante o processo de verificação. No entanto, os tipos, as funções e os lemas construídos para suportar e converter as duas representações facilitaram bastante o processo de verificação.

5.3 Processo de verificação do modelo B assembly

Esta seção descreve o processo de verificação do modelo B *assembly* *TestCalc_basm* e o efeito de uma das técnicas utilizadas. O modelo *TestCalc_basm* continha 200 obrigações de prova do tipo EBD (Expressões Bem Definidas) e 568 não óbvias. As obrigações de prova EBD foram rapidamente resolvidas através de comandos de prova e do provador interativo. As outras 568 não foram resolvidas tão facilmente. Felizmente, existe um comando de prova extremamente eficiente que agiliza a verificação, pois esse comando de prova é reusável em obrigações de prova semelhantes, as quais são comuns na modelagem B *assembly*.

Este comando de prova praticamente resolve todas as obrigações de prova as quais representam um fluxo de execução inválido, isto é, no caso do modelo *TestCalc_basm* representam a possibilidade de fluxos de execução não lineares. Essas obrigações de prova são criadas porque o gerador de obrigações de prova constrói fórmulas para avaliar todas as posibilidades do fluxo de execução do modelo. Como o modelo especificado tem um fluxo de execução linear, ou seja, o contador de programa (*pc*) efetua apenas saltos para a instrução seguinte, então o número de possibilidades do fluxo de execução é mínimo. Nesse caso, o comando de prova citado ajudou a verificar rapidamente 465 obrigações de prova (81%). Um fato ainda mais interessante é que foi utilizado um *time out* de um segundo para resolver cada obrigação de prova. Dessa forma, foi necessário no máximo 465 segundos para resolver 81% das obrigações de prova. As 103 demais obrigações de prova foram resolvidas interativamente, pois essas eram mais complexas[3]. Para esse comando de prova funcionar é importante declarar no invariante do *while* de cada instrução os possíveis próximos valores do contador de programa. Esse e outros comandos de prova podem ser consultados no anexo deste trabalho.

[3]Nenhuma das 103 obrigações de prova foi resolvida no provador automático na força zero e com um *time out* de 180 segundos.

Nesse modelo os passos das provas interativas seguem alguns padrões, os quais foram identificados pelo projetista e isso ajudou bastante a verificação. Contudo, o projetista ainda não conseguiu definir uma sequência de comandos de prova suficientemente genérica para resolver várias obrigações de prova entre as 103 não resolvidas automaticamente. Finalmente, a construção de um invariante válido e a realização das provas interativas do modelo *TestCal_basm* apresentado anteriormente são as atividades mais complexas do processo de verificação, já que essas duas atividades consomem aproximadamente 70% e 80% de todo o processo de especificação e verificação.

5.4 Simulação do código assembly

A simulação do *software*[4] tem um papel importante para analisar seu comportamento, pois, no simulador, é possível avaliar e manipular os valores dos registradores, da memória e das portas de entrada e saída. O uso do simulador permite ganhar confiança no programa *assembly* desenvolvido, assim como no próprio modelo formal do conjunto de instruções do Z80, visto que o manual possui pequenos erros.

Por conseguinte, do ponto de vista do sistema implantado em campo, o Z80 recebe do radar ou do sistema supervisório o valor do nível do tanque no início da fase de drenagem. E ao término dessa fase, o Z80 recebe em outra porta o valor do nível final do tanque. Em seguida, o Z80 deve calcular e disponibilizar em outras duas portas o fator de água livre e o do óleo bruto produzido.

Esse código *assembly* foi simulado com o aplicativo [Soso 2002]. A Figura 14 ilustra o final da execução do *software* e os valores dos elementos do microcontrolador. Todos esses valores do simulador são representados no sistema de numeração hexadecimal e as portas de entrada e saída, também no sistema binário. A janela do lado superior esquerdo contém os estados dos registradores do microcontrolador e as opções para controlar a execução do programa. A janela superior do lado direito contém o código *assembly* em execução e uma seta amarela indicando a posição atual do contador de programa. A janela inferior esquerda contém um editor da memória de dados do microcontrolador. A janela inferior direita contém as informações que estão representadas nas portas de entrada e saída. A porta do endereço *00H* contém o valor inicial do nível do tanque (10); a do endereço *01H*, o valor final do nível do tanque (2); a do enderço *02H*, o fator de proporção de água livre produzido (8) e do endereço *03H*, o fator de proporção de óleo bruto produzido (2). O leitor pode conferir como cada elemento ilustrado na Figura 14 (resgistradores, portas de entrada e saída, interrupções e memória) foi especificado consultando o anexo deste trabalho.

[4] A sequência de instruções representadas no modelo *TestCalc_basm* formam um *software* em linguagem *assembly* do Z80, o qual é ilustrado no lado superior direito da Figura 14.

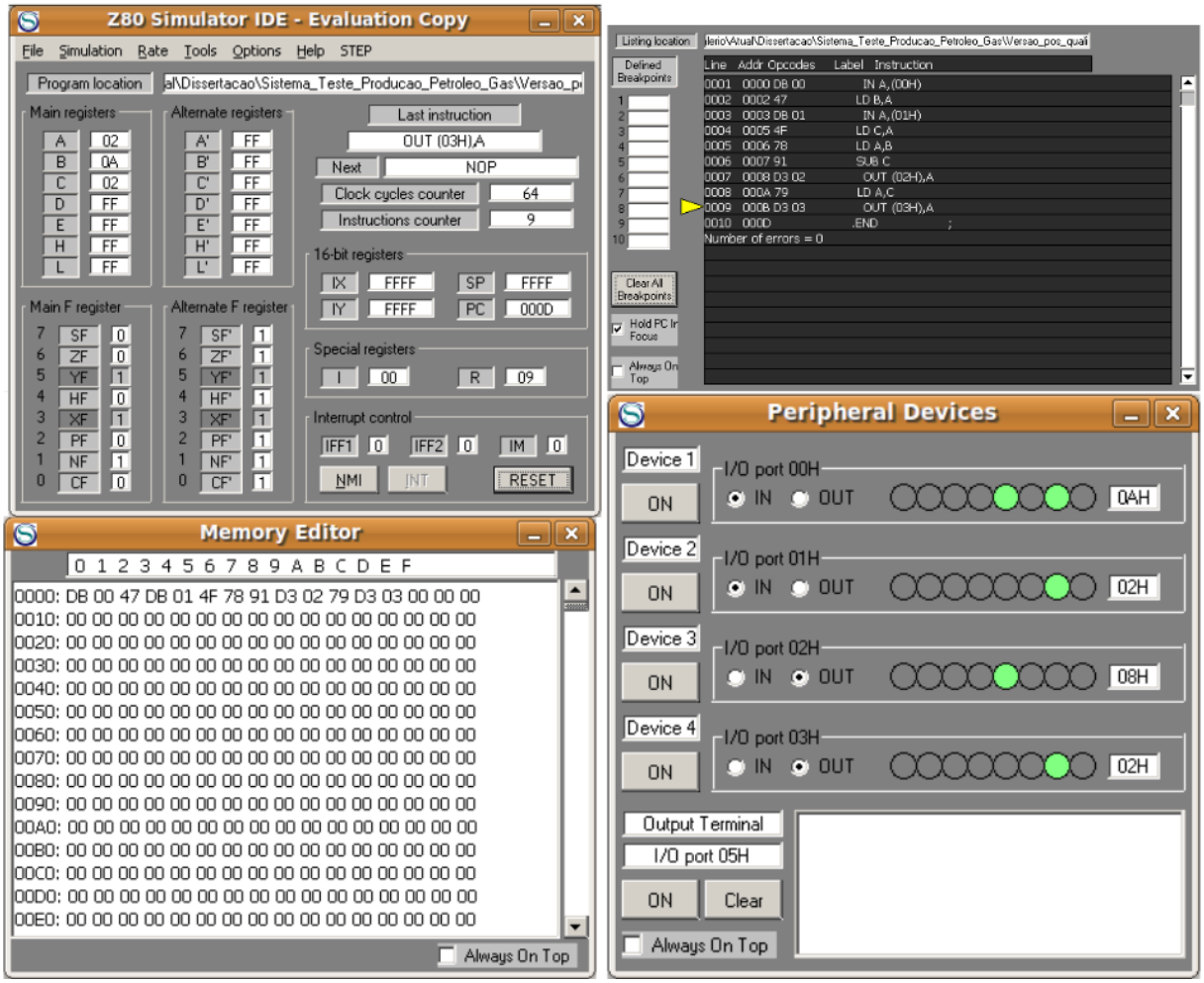

Figura 14: Janelas do simulador [Soso 2002].

5.5 Considerações finais sobre o estudo de caso

O modelo *TestCalc_basm* foi o primeiro construído de acordo com a proposta de [Dantas et al. 2008] utilizando as bibliotecas de *hardware* e tipos inteiros. Essas bibliotecas contêm tipos, funções e lemas que possibilitaram estabelecer a relação entre o modelo B *assembly* e o modelo mais abstrato. Essas definições mantiveram a assinatura das operações do modelo abstrato com relação ao B *assembly*, simplificaram o processo de verificação e tornaram mais fiel a representação do microcontrolador, tudo isso sem a necessidade de estender o método B.

A elaboração desse modelo foi muito importante para amadurecer o processo de verificação, pois a construção e verificação do modelo ajudou ao projetista adquirir experiências com o provador interativo e a descobrir comandos de prova que reduzem significativamente o tempo do processo de prova. Essa experiência torna mais viável a especificação de outros *softwares* mais complexos, pois a maioria dos comandos de provas podem ser usados novamente em obrigações de prova similares e o padrão de especificação utilizado ajuda na modelagem de novas plataformas.

Finalmente, o funcionamento do *software* desenvolvido como um sistema redudante deve oferecer uma melhor confiança e qualidade, pois o engenheiro responsável terá mais um meio de comparação dos testes de produção e mais garantias. Essas garantias devem-se à existência de um sistema redundante e a sua verificação formal até um dos últimos níveis de representação do *software*. Logo, o sistema pode também melhorar a eficácia dos testes, visto que menos testes falhos podem existir.

Conclusões

Este trabalho provê contribuições para estabelecer uma abordagem inovadora para resolver o problema da geração de *software* correto por construção e, assim, contribuir à resolução de um dos grandes desafios da computação em pesquisa no Brasil e no mundo. Nesse sentido, vários estudos foram realizados para garantir o embasamento teórico, melhorar a viabilidade da metodologia proposta e avançar na atual linha de pesquisa. O início da pesquisa no nicho dos microcontroladores e microprocessadores foi uma decisão muito importante, pois esses dispositivos são simples, possuem um mercado importante e evitam inicialmente a preocupação de modelar aspectos extremamente complexos, o que é comum nos modernos processadores. Dessa forma, a pesquisa focaliza os esforços na essência da verificação em nível de *assembly* e especificação do conjunto de instruções *assembly*, o que produz resultados mais rapidamente. Os trabalhos realizados aproveitam bastante da maturidade e eficiência das ferramentas e práticas do método B. Essa maturidade tem sido adquirida por vários anos com a modelagem de sistemas reais e de porte industrial, o que fornece uma boa credibilidade à pesquisa. Além disso, as ferramentas do método B suportam funcionalidades avançadas, as quais possibilitam a partir do modelo formal: simular sua execução, gerar documentação em código LaTeX e código na linguagem C, aplicar técnicas avançadas de prova e até paralelizar o processo de prova em uma rede de computadores. Essas técnicas avançadas de verificação são úteis para o projeto de um arcabouço de compilação formal.

Aproveitando todas as potencialidades citadas e a proposta desta dissertação, o presente trabalho demonstrou que o método B é capaz de documentar, simular, modelar e verificar instruções *assembly* de microcontroladores e microprocessadores. Pois nesse trabalho, foi possível identificar erros e ambiguidades no manual oficial do microcontrolador modelado. Adicionalmente, um pequeno *software* da área do petróleo foi desenvolvido e verificado até o nível *assembly*, utilizando a especificação modelada do Z80. Esses trabalhos resultaram em exemplos interessantes que podem ser seguidos para verificação de outras plataformas e modelos de *software* até a linguagem *assembly*.

Alguns dos trabalhos desenvolvidos ajudam a esclarecer outros detalhes da proposta dessa linha de pesquisa. Eles foram publicados em diferentes eventos e são citados a seguir de acordo com a ordem cronológica de publicação.

SEMISH 2008 - [Dantas et al. 2008] apresenta a metodologia de desenvolvimento formal até

o nível de *assembly* como um dos grandes desafios da Sociedade Brasileira de Computação.

SBMF 2008 - [Dantas et al. 2008] apresenta mais detalhes sobre a proposta de verificação em nível de *assembly* usando o método B e um pequeno exemplo de *software* verificado em 3 diferentes plataformas .

ERMAC 2008 - [Medeiros, Galvão e Déharbe 2008] descreve as bibliotecas de *hardware* e tipos inteiros de 8 e 16 bits e a modelagem inicial do Z80.

SBMF 2009 - [Medeiros e Déharbe 2009] mostra uma visão geral da modelagem completa do Z80 e alguns detalhes sobre seu processo de verificação.

Esses trabalhos concretizam um passo importante para verificação de *software* até a linguagem *assembly*, apresentam as dificuldades atuais e sugerem melhorias nas ferramentas de suporte B. O presente trabalho preocupa-se com consistência do código *assembly* gerado após a aplicação do método B, o que não está no escopo das mais importantes ferramentas de suporte B [Clearsy 2009, B-Core 1995, Leuschel e Butler 2003]. A introdução de erros no código *assembly* é um acontecimento raro e de difícil diagnóstico, principalmente porque a depuração de *software* na linguagem *assembly* é muito complexa. Contudo, um fato mais comum que provoca a mudança da semântica do *software* é quando acontece a alteração na versão do compilador. Portanto, a escolha da versão do compilador deve ser mantida em projetos críticos para evitar eventuais problemas. Logo, um cuidado especial deve ser tomado com o processo de transformação de código para a linguagem *assembly*, principalmente nos compiladores de uso restrito. No entanto, para realizar esse cuidado adicional é necessária a construção de modelos formais da plataforma e do *software*. Este trabalho desenvolveu exemplos desses modelos e isto implicou na verificação de um grande número de obrigações de provas (4429). Essa atividade consumiu um longo e exaustivo tempo de trabalho, pois algumas vezes foi necessário corrigir o modelo e refazer as provas, o que poderia ser evitado se a ferramenta aproveitasse melhor as provas realizadas anteriormente. Esses fatos demonstram a importância da indústria e academia proverem novas funcionalidades e melhorias nas tecnologias de verificação para o método B.

Como trabalhos futuros, o autor pretende especificar e verificar outros estudos de casos a fim de modelar estruturas de dados mais avançadas, consequentemente proporcionar novas experiências no processo de verificação. Existe também a pretensão de especificar as instruções *assembly* de uma plataforma genérica intermediária, por exemplo, LLVM [Adve et al. 2003]. As experiências já realizadas e as futuras devem fornecer conhecimentos e técnicas úteis para construção de um compilador formal. Além disso, o autor deste trabalho vislumbra o desenvolvimento de um arcabouço formal de refinamentos pré-verificados, ou pelo menos parcialmente, como existe para outros formalismos.

Os resultados desse trabalho demonstram a viabilidade da proposta de desenvolvimento

formal até o nível de *assembly*, pelo menos para sistemas de pequeno porte. Esses resultados mostram também que as linguagens de especificações formais para *software* podem ser usadas para definir a semântica do conjunto de instruções de microcontroladores e microprocessadores. Finalmente, a continuação da linha de pesquisa desse trabalho pode colaborar efetivamente na solução dos grandes desafios da computação no Brasil e no mundo [Dantas et al. 2008].

Referências

[Abrial 1996]ABRIAL, J. R. *The B Book: Assigning Programs to Meanings*. 1. ed. United States of America: Cambridge University Press, 1996. (1, 1).

[Abrial 2007]ABRIAL, J.-R. A system development process with event-b and the rodin platform. In: BUTLER, M.; HINCHEY, M. G.; LARRONDO-PETRIE, M. M. (Ed.). *Formal Methods and Software Engineering*. Berlin: Springer, 2007. (Lecture Notes in Computer Science, v. 4789), p. 1–3. ISBN 978-3-540-76648-3.

[Adve et al. 2003]ADVE, V. et al. LLVA: A Low-level Virtual Instruction Set Architecture. In: *Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36)*. San Diego, California: [s.n.], 2003.

[Aljer et al. 2003]ALJER, A. et al. Bhdl: Circuit design in B. In: . *ACSD*. France, 2003. p. 241–242.

[B-Core 1995]B-CORE. Tapsoft'95: Theory and practice of software development, 6th international joint conference caap/fase, aarhus, denmark, may 22-26, 1995, proceedings. In: ICFTAPSOFT'95EM. *TAPSOFT*. Berlin, 1995. p. 805–806.

[B-Core 1999]B-CORE. *A Comparison of Z and VDM with B/AMN*. agos 1999. Disponível em: http://www.b-core.com/ZVdmB.html. Acessado em: 18 abr 2007.

[Bertot 2004]BERTOT, P. C. Y. *Interactive Theorem Proving and Program Development: Coq'Art: the Calculus of Inductive Constructions* . 1. ed. [S.l.]: Springer, 2004. (1, 1).

[Blazy, Dargaye e Leroy 2006]BLAZY, S.; DARGAYE, Z.; LEROY, X. Formal verification of a c compiler front-end. In: *FM*. [S.l.: s.n.], 2006. p. 460–475.

[Burkhardt 2000]BURKHARDT, R. An ISO standard guards the Ada Hen House. In: IEEE. *ADA*. United States of America, 2000. Disponível em: http://ieeexplore.ieee.org/xplore/login.jsp?url=/iel5/52/17783/00820017.pdf Acessado em:18 abr 2007.

[Clearsy 2009]CLEARSY. *Atelier B web site*. 2009.
Disponível em: http://www.atelierb.eu. Acesso em: 04 abril 2009.

[Cury 2007]CURY, E. *Certificação de Software*. 2007. Disponível em:http://www.aviacaocivil.ifi.cta.br/Cursos/SeminarioRCF_2003. Acessado em: 14 de mai 2007.

[Dantas et al. 2008]DANTAS, B. P. et al. Applying the B method to take on the grand challenge of verified compilation. In: SBMF. *Brazilian Symposium on Formal Methods*. Salvador - BA, 2008.

[Dantas et al. 2008]DANTAS, B. P. et al. Proposta e avaliação de uma abordagem de desenvolvimento de software fidedigno por construção com o método b. In: XXXV SEMISH. *Seminário Integrado de Software e Hardware*. Belém - PA, 2008.

[Dondossola 1999]DONDOSSOLA, G. Formal methods for the engineering and certification of safety-critical knowledge-based systems. In: VERMESAN, A. I.; COENEN, F. (Ed.). *EUROVAV*. [S.l.]: Kluwer, 1999. p. 113–129. ISBN 0-7923-8645-0.

[Evans e Grant 2008]EVANS, N.; GRANT, N. Towards the formal verification of a java processor in event-b. *Electr. Notes Theor. Comput. Sci.*, v. 201, p. 45–67, 2008.

[Gajski e Vahid 1995]GAJSKI, D. D.; VAHID, F. Specification and design of embedded hardware-software systems. *IEEE Design & Test of Computers*, v. 12, n. 1, p. 53–67, 1995.

[Hartel e Moreau 2001]HARTEL, P. H.; MOREAU, L. Formalizing the safety of Java, the Java virtual machine, and Java card. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 33, n. 4, p. 517–558, 2001. ISSN 0360-0300.

[Hoare 2005]UK COMPUTING RESEARCH COMMITTEE. *The Verifying Compiler: A GrandChallenge for Computing Research*. UK: Workshop on Grand Challenges for Computing Research, 2005. Disponível em:http://www.nesc.ac.uk/esi/events/Grand_Challenges/workshop02.html Acessado em:18 abr 2007.

[Leinenbach, Paul e Petrova 2005]LEINENBACH, D.; PAUL, W. J.; PETROVA, E. Towards the formal verification of a c0 compiler: Code generation and implementation correctnes. In: *SEFM*. [S.l.: s.n.], 2005. p. 2–12.

[Leroy 2006]LEROY, X. Formal certification of a compiler back-end. *INRIA Rocquencourt*, 2006. Disponível em:http://pauillac.inria.fr/ xleroy/publi/compiler-certif.pdf Acessado em : 14 de mai 2007.

[Leroy 2006]LEROY, X. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In: INRIA. *POPL*. France, 2006. p. 42–54.

[Leuschel 2008]LEUSCHEL, M. Towards demonstrably correct compilation of java byte code. In: BOER, F. S. de; BONSANGUE, M. M.; MADELAIN, E. (Ed.). *FMCO*. [S.l.]: Springer, 2008. (Lecture Notes in Computer Science, v. 5751), p. 119–138. ISBN 978-3-642-04166-2.

[Leuschel e Butler 2003]LEUSCHEL, M.; BUTLER, M. ProB: A model checker for B. In: ARAKI, K.; GNESI, S.; MANDRIOLI, D. (Ed.). *FME 2003: Formal Methods*. Pisa, Italy: Springer, 2003. (LNCS 2805), p. 855–874. ISBN 3-540-40828-2.

[Lima 2000]LIMA, C. E. G. *Automação de Testes de Produção e Determinação de BS&W de poços produtores de petróleo*. Dissertação (Mestrado) — UFRN, Natal, 2000.

[Ludovic e Lanet 1999]LUDOVIC, C.; LANET, J. L. *A Formal Specification of the Java Bytecode Semantics using the B method*. France, 1999.

[Marinho et al. 2008]MARINHO, E. S. et al. A Ferramenta Batcave para a Verificação de Especificações Formais na Notação B. In: SIMPóSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE (SBES). *Sessão de Ferramentas*. Campinas, SP, 2008.

[Medeiros 2007]MEDEIROS, V. G. de. *Aplicação do método B para a construção de programas assembly*. 2007. Relatório de graduação - Dimap. Natal/RN.

[Medeiros e Déharbe 2009]MEDEIROS, V. G. J.; DéHARBE, D. Formal Modelling of a Microcontroller Instruction Set in B. In: SBMF. *Brazilian Symposium on Formal Methods - Student Paper*. Gramado, RS, 2009.

[Medeiros, Galvão e Déharbe 2008]MEDEIROS, V. G. J.; GALVãO, S. S. L.; DéHARBE, D. Modelagem de microcontroladores em B. In: ENCONTRO REGIONAL DE MATEMáTICA APLICADA E COMPUTACIONAL. *Anais do VIII ERMAC*. Natal - RN, 2008.

[Morgan 1989]MORGAN, C. Types and invariants in the refinement calculus. In: SNEPSCHEUT, J. L. A. van de (Ed.). *MPC*. Amsterdam: Springer, 1989. (Lecture Notes in Computer Science, v. 375), p. 363–378. ISBN 3-540-51305-1.

[Ramasamy, Cukier e Sanders 2003]RAMASAMY, H. V.; CUKIER, M.; SANDERS, W. H. *Formal Verification of an Intrusion-Tolerant Group Membership Protocol*. 2003.

[Schneider 2001]SCHNEIDER, S. *The B-method an introduction*. 1. ed. Great Britain: PALGRAVE MACMILLAN, 2001. (1, 1).

[Silva 2008]SILVA, P. S. e. *Automação Da Drenagem No Teste De Produção Convencional Em Tanque Cilíndrico*. Dissertação (Mestrado) — UFRN, 2008.

[Soso 2002]SOSO, V. *Z80 Simulator IDE*. 2002. On-line. Disponível em:http://www.oshonsoft.com Acessado em:18 abr 2009.

[Velev 2004]VELEV, M. N. Efficient formal verification of pipelined processors with instruction queues. In: *ACM Great Lakes Symposium on VLSI*. [S.l.: s.n.], 2004. p. 92–95.

[Wordsworth 1996]WORDSWORTH, J. *Software Engineering with B*. England: Addison-Wesley, 1996. 331p.

[Young 2003]YOUNG, S. *The Undocumented Z80 Documented*. [S.l.], November 2003. Disponível em: http://www.myquest.nl/z80undocumented/z80-documented.pdf. Acessado em:18 abr 2007.

[Zilog 2001]ZILOG. *Z80 Family CPU User Manual*. 910 E. Hamilton Avenue, 2001. Disponível em:http://www.zilog.com/docs/z80/um0080.pdf. Acessado em:18 abr 2007.

6 Anexos

6.1 Comandos de prova

O uso de comandos de prova pode acelerar bastante o processo de prova, portanto dois comandos de provas simples, eficazes, eficientes e muito úteis são apresentados. Quando esses dois comandos de provas são utilizados é recomendado ajustar o *time out* do provador para 1 segundo, o que é suficiente para resolver a grande maioria das obrigações de prova e isso evita que o provador desperdice muito tempo desnecessariamente em obrigações de prova que não são resolvidas por estes comandos.

O comando de prova seguinte é utilizado para resolver a maioria das obrigações de prova EBD do modelo do Z80. Esse comando de prova é aplicado nas obrigações de prova que satisfazem o padrão: $x \in dom(y)$. A sequência de comandos: seleciona a força mínima do provador, remove as hipóteses óbvias da pilha, substitui o termo $dom(y)$ por um equivalente, executa o simplificador e o provador automático.

```
Pattern( x ∈ dom(y) ) & ff(0) & dd & eh(dom(y)) & ss & pr
```

O comando de prova definido a seguir foi utilizado para resolver 81% das obrigações de prova do modelo *TestCal_basm*. A sequência de comandos: seleciona a força mínima do provador e executa o provador tático com o conjunto de regras *ContradictionXY*.

```
ff(0) & pr(Tac(ContradictionXY))
```

6.2 Invariante do modelo

A especificação completa do invariante do modelo *TestCal_basm* é apresentada a seguir, essa especificação foi citada no capítulo 5.

INVARIANT

$$
\begin{aligned}
& local\_pc \in 0 \,..\, 9 \wedge rgs8 \in id\_reg\_8 \rightarrow BYTE \\
& \wedge\ r\_ \quad \in BYTE \wedge io\_ports \in BYTE \rightarrow BYTE \\
& \wedge\ pc \in 0 \,..\, 9 \wedge free\_water\_factor \in UCHAR \wedge \\
& (local\_pc = 0 \Rightarrow (\ pc = 0 \wedge instruction\_next(pc) = 1 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ ) \wedge \\
& (local\_pc = 1 \Rightarrow (\ instruction\_next(pc) = 2 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
& \qquad byte\_uchar(\ rgs8(a0)\ ) = (\ initial\_level\ ) \quad \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ )\ ) \wedge \\
& (local\_pc = 2 \Rightarrow (\ instruction\_next(pc) = 3 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
& \qquad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
& \qquad byte\_uchar(\ rgs8(a0)\ ) = (\ initial\_level\ ) \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ )\ ) \wedge \\
& (local\_pc = 3 \Rightarrow (\ instruction\_next(pc) = 4 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
& \qquad byte\_uchar(\ rgs8(a0)\ ) = (\ final\_level\ ) \wedge \\
& \qquad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ )\ ) \wedge \\
& (local\_pc = 4 \Rightarrow (\ instruction\_next(pc) = 5 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
& \qquad byte\_uchar(\ rgs8(a0)\ ) = (\ final\_level\ ) \wedge \\
& \qquad byte\_uchar(\ rgs8(c0)\ ) = (\ final\_level\ ) \wedge \\
& \qquad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ )\ ) \wedge \\
& (local\_pc = 5 \quad \Rightarrow (\ instruction\_next(pc) = 6 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
& \qquad byte\_uchar(\ rgs8(a0)\ ) = (\ initial\_level\ ) \wedge \\
& \qquad byte\_uchar(\ rgs8(c0)\ ) = (\ final\_level\ ) \wedge \\
& \qquad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ )\ ) \wedge \\
& (local\_pc = 6 \quad \Rightarrow (\ instruction\_next(pc) = 7 \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
& \qquad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
& \qquad byte\_uchar(\ rgs8(a0)\ ) = (\ (initial\_level - final\_level) \ \mathbf{mod}\ 256) \wedge \\
& \qquad byte\_uchar(\ rgs8(c0)\ ) = (\ final\_level\ ) \wedge
\end{aligned}
$$

$$
\begin{array}{l}
\quad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(2))) = \ free\_water\_factor \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ )\ ) \wedge \\
(local\_pc = 7 \quad \Rightarrow (\ instruction\_next(pc) = 8 \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
\quad byte\_uchar(\ rgs8(a0)\ ) = (\ (initial\_level - final\_level) \ \mathbf{mod}\ 256) \wedge \\
\quad byte\_uchar(\ rgs8(c0)\ ) = (\ final\_level\ ) \wedge \\
\quad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(2))) = \ ((initial\_level - final\_level) \ \mathbf{mod}\ 256) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor \wedge \\
(local\_pc = 8 \quad \Rightarrow (\ instruction\_next(pc) = 9 \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
\quad byte\_uchar(\ rgs8(a0)\ ) = (\ final\_level) \wedge \\
\quad byte\_uchar(\ rgs8(c0)\ ) = (\ final\_level\ ) \wedge \\
\quad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(2))) = \ ((initial\_level - final\_level) \ \mathbf{mod}\ 256) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(3))) = \ oil\_factor\ ) \wedge \\
(local\_pc = 9 \quad \Rightarrow (\ pc = 9 \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(0))) = \ initial\_level \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(1))) = \ final\_level \wedge \\
\quad byte\_uchar(\ rgs8(a0)\ ) = (\ final\_level) \wedge \\
\quad byte\_uchar(\ rgs8(c0)\ ) = (\ final\_level\ ) \wedge \\
\quad byte\_uchar(\ rgs8(b0)\ ) = (\ initial\_level\ ) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(2))) = \ ((initial\_level - final\_level) \ \mathbf{mod}\ 256) \wedge \\
\quad byte\_uchar(io\_ports(uchar\_byte(3))) = \ final\_level\ ))
\end{array}
$$

6.3 Modelagem das instruções do Z80

As instruções do Z80 são apresentadas a seguir. O leitor interessando em maiores detalhes pode consultar todos os modelos no repositório do projeto em: http://code.google.com/p/b2asm

MACHINE
Z80
INCLUDES
MEMORY
SEES
ALU ,
BIT_DEFINITION,
BIT_VECTOR_DEFINITION,
BYTE_DEFINITION,
BV16_DEFINITION,

UCHAR_DEFINITION,
SCHAR_DEFINITION,
SSHORT_DEFINITION,
USHORT_DEFINITION,
POWER2

SETS

$id\_reg\_8 = \{\ a0\ ,\ f0\ ,\ f\_0\ ,\ a\_0\ ,$
$b0\ ,\ c0\ ,\ b\_0\ ,\ c\_0\ ,$
$d0\ ,\ e0\ ,\ d\_0\ ,\ e\_0\ ,$
$h0\ ,\ l0\ ,\ h\_0\ ,\ l\_0\ \}\ ;$
$id\_reg\_16 = \{\ BC\ ,\ DE\ ,\ HL\ ,\ SP\ ,\ AF\ \}$

ABSTRACT_VARIABLES

$rgs8,$
$pc\ ,\ sp\ ,\ ix\ ,\ iy\ ,$
$i\_,r\_\ ,$
$iff1\ ,\ iff2,$
$im\ ,$
io_ports

INVARIANT

$rgs8 \in id\_reg\_8 \rightarrow BYTE \wedge$
$pc \in USHORT \wedge sp \in BV16 \wedge ix \in BV16 \wedge iy \in BV16 \wedge$
$i\_ \in BYTE \wedge r\_ \quad \in BYTE \wedge$
$iff1 \in BIT \wedge \quad iff2 \in BIT \wedge$
$im : (BIT \times BIT) \wedge$
$io\_ports \quad \in BYTE \rightarrow BYTE$

DEFINITIONS

$bv\_BC == byte\_bv16\ (\ rgs8\ (\ b0\ )\ ,\ rgs8\ (\ c0\ )\ )\ ;$
$bv\_HL == byte\_bv16\ (\ rgs8\ (\ h0\ )\ ,\ rgs8\ (\ l0\ )\ )\ ;$
$bv\_DE == byte\_bv16\ (\ rgs8\ (\ d0\ )\ ,\ rgs8\ (\ e0\ )\ )\ ;$
$bv\_AF == byte\_bv16\ (\ rgs8\ (\ a0\ )\ ,\ rgs8\ (\ f0\ )\ )\ ;$

$bv\_9BC0 == mem\ (\ byte\_bv16\ (\ rgs8\ (\ b0\ )\ ,\ rgs8\ (\ c0\ )\ )\ );$
$bv\_9DE0 == mem\ (\ byte\_bv16\ (\ rgs8\ (\ d0\ )\ ,\ rgs8\ (\ e0\ )\ )\ );$
$bv\_9HL0 == mem\ (\ byte\_bv16\ (\ rgs8\ (\ h0\ )\ ,\ rgs8\ (\ l0\ )\ )\ );$
$bv\_9AF0 == mem\ (\ byte\_bv16\ (\ rgs8\ (\ a0\ )\ ,\ rgs8\ (\ f0\ )\ )\ );$

$bv\_9SP0 == mem\ (\ sp\ );$
$bv\_9IX0 == mem\ (\ ix\ );$
$bv\_9IY0 == mem\ (\ iy\ );$

$sp\_plus\_one == ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ bv16\_ushort\ (\ sp\ )\ ,\ 1\ )\ );$
$sp\_plus\_two == ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ bv16\_ushort\ (\ sp\ )\ ,\ 2\ )\ );$
$sp\_minus\_one == ushort\_bv16\ (\ sub16USHORT\ (\ 0\ ,\ bv16\_ushort\ (\ sp\ )\ ,\ 1\ )\ );$
$sp\_minus\_two == ushort\_bv16\ (\ sub16USHORT\ (\ 0\ ,\ bv16\_ushort\ (\ sp\ )\ ,\ 2\ )\ );$

$z\_s \quad == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 7\ )\ ;$
$z\_z \quad == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 6\ )\ ;$
$z\_00 == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 5\ )\ ;$
$z\_h \quad == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 4\ )\ ;$
$z\_01 == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 3\ )\ ;$
$z\_p \quad == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 2\ )\ ;$
$z\_n \quad == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 1\ )\ ;$
$z\_c \quad == bitget\ (\ rgs8\ (\ f0\ )\ ,\ 0\ )$

CONCRETE_CONSTANTS

$get\_bv\_reg16\ ,$
$REG16\_TO\_REG8\ ,$
$REG8\_TO\_REG16\ ,$

$update\_flag\_reg$,

$bv\_ireg\_plus\_d$,
$bv\_9ireg\_plus\_d0$,

cc_get

PROPERTIES

$get\_bv\_reg16 : (BV16 \times (id\_reg\_8 \rightarrow BYTE) \times id\_reg\_16) \rightarrow (BV16)$
$\wedge \forall (sp\_ , rgs8\_ , r1) .$
$(sp\_ \in BV16 \wedge rgs8\_ : (id\_reg\_8 \rightarrow BYTE) \wedge r1 \in id\_reg\_16$
$\Rightarrow (r1 = BC \Rightarrow get\_bv\_reg16 (sp\_ , rgs8\_ , r1) = byte\_bv16 (rgs8\_ (b0) , rgs8\_ (c0))) \wedge$
$(r1 = DE \Rightarrow get\_bv\_reg16 (sp\_ , rgs8\_ , r1) = byte\_bv16 (rgs8\_ (d0) , rgs8\_ (e0))) \wedge$
$(r1 = HL \Rightarrow get\_bv\_reg16 (sp\_ , rgs8\_ , r1) = byte\_bv16 (rgs8\_ (h0) , rgs8\_ (l0))) \wedge$
$(r1 = SP \Rightarrow get\_bv\_reg16 (sp\_ , rgs8\_ , r1) = sp\_) \wedge$
$(r1 = AF \Rightarrow get\_bv\_reg16 (sp\_ , rgs8\_ , r1) = byte\_bv16 (rgs8\_ (a0) , rgs8\_ (f0)))$
$) \wedge$

$REG16\_TO\_REG8 \in id\_reg\_16 \rightarrow (id\_reg\_8 \times id\_reg\_8) \wedge$
$REG16\_TO\_REG8 (BC) = (b0 , c0) \wedge$
$REG16\_TO\_REG8 (DE) = (d0 , e0) \wedge$
$REG16\_TO\_REG8 (HL) = (h0 , l0) \wedge$
$REG16\_TO\_REG8 (AF) = (a0 , f0) \wedge$

$REG8\_TO\_REG16 : (id\_reg\_8 \times id\_reg\_8) \nrightarrow id\_reg\_16 \wedge$
$REG8\_TO\_REG16 = REG16\_TO\_REG8 ^{-1} \wedge$

$update\_flag\_reg : (BIT \times BIT \times BIT \times BIT \times BIT \times BIT \rightarrow (\{f0\} \times BYTE)) \wedge$
$update\_flag\_reg = \boldsymbol{\lambda} (s7 , z6 , h4 , pv2 , n\_add\_sub , c0) .$
$(s7 \in BIT \wedge z6 \in BIT \wedge h4 \in BIT \wedge pv2 \in BIT \wedge n\_add\_sub \in BIT \wedge c0 \in BIT |$
$(f0 \mapsto [c0, n\_add\_sub, pv2, 1, h4, 1, z6, s7])) \wedge$

$bv\_ireg\_plus\_d : (BV16 \times SCHAR \rightarrow BV16) \wedge$
$bv\_ireg\_plus\_d = \boldsymbol{\lambda} (ix\_iy , desloc) . (ix\_iy \in BV16 \wedge desloc \in SCHAR |$
$ushort\_bv16 ((bv16\_ushort (ix\_iy) + desloc) \mathbf{mod} 65536)) \wedge$

$bv\_9ireg\_plus\_d0 : ((BV16 \rightarrow BYTE) \times BV16 \times SCHAR) \rightarrow BYTE \wedge$
$bv\_9ireg\_plus\_d0 = \boldsymbol{\lambda} (mem , ix\_iy , desloc) . (mem : (BV16 \rightarrow BYTE) \wedge ix\_iy \in BV16 \wedge desloc \in SCHAR |$
$mem (bv\_ireg\_plus\_d(ix\_iy , desloc))) \wedge$

$cc\_get : ((id\_reg\_8 \rightarrow BYTE) \times (0 \mathbf{..} 8)) \rightarrow BIT \wedge$
$\forall (rgs8\_).(rgs8\_ \in id\_reg\_8 \rightarrow BYTE \Rightarrow cc\_get(rgs8\_ , 0) = 1 - bitget (rgs8\_ (f0) , 6) \wedge$
$cc\_get(rgs8\_ , 1) = bitget (rgs8\_ (f0) , 6) \wedge$
$cc\_get(rgs8\_ , 2) = 1 - bitget (rgs8\_ (f0) , 0) \wedge$
$cc\_get(rgs8\_ , 3) = bitget (rgs8\_ (f0) , 0) \wedge$
$cc\_get(rgs8\_ , 4) = 1 - bitget (rgs8\_ (f0) , 2) \wedge$
$cc\_get(rgs8\_ , 5) = bitget (rgs8\_ (f0) , 2) \wedge$
$cc\_get(rgs8\_ , 6) = 1 - bitget (rgs8\_ (f0) , 7) \wedge$
$cc\_get(rgs8\_ , 7) = bitget (rgs8\_ (f0) , 7)$)

ASSERTIONS

$\mathbf{ran} (mem) \subseteq BYTE \wedge$
$\mathbf{dom} (mem) = BV16 \wedge$

$\mathbf{ran}\ (\ rgs8\ ) \subseteq BYTE \wedge$
$\mathbf{dom}\ (\ rgs8\ ) = id\_reg\_8 \wedge$

$instruction\_next(pc) \in USHORT \wedge$

$bv\_9BC0 \in BYTE \wedge mem\ (\ byte\_bv16\ (\ schar\_byte\ (\ 0\ )\ ,\ bv\_9BC0\ )\ ) \in BYTE \wedge$
$bv\_9DE0 \in BYTE \wedge mem\ (\ byte\_bv16\ (\ schar\_byte\ (\ 0\ )\ ,\ bv\_9DE0\ )\ ) \in BYTE \wedge$
$bv\_9HL0 \in BYTE \wedge mem\ (\ byte\_bv16\ (\ schar\_byte\ (\ 0\ )\ ,\ bv\_9HL0\ )\ ) \in BYTE \wedge$
$bv\_9AF0 \in BYTE \wedge mem\ (\ byte\_bv16\ (\ schar\_byte\ (\ 0\ )\ ,\ bv\_9AF0\ )\ ) \in BYTE \wedge$

$bv\_9SP0 \in BYTE \wedge$
$bv\_9IX0 \in BYTE \wedge$
$bv\_9IY0 \in BYTE \wedge$

$update\_refresh\_reg(r\_) \in BYTE \wedge$

$\mathbf{dom}(update\_flag\_reg) = (\ BIT \times BIT \times BIT \times BIT \times BIT \times BIT\ ) \wedge$
$\mathbf{ran}(update\_flag\_reg) \in \mathcal{P}\ (\{f0\} \times BYTE) \wedge$

$\forall\ (b1,b2,b3,b4,b5,b6).(\ b1 \in BIT \wedge b2 \in BIT \wedge b3 \in BIT \wedge b4 \in BIT \wedge b5 \in BIT \wedge b6 \in BIT \Rightarrow update\_flag\_reg(b1,b2,b3,b4,b5,b6) \in \{f0\} \times BYTE\ ) \wedge$

$\forall\ (xx).(xx \in id\_reg\_8 \rightarrow BYTE \Rightarrow (rgs8 \lhd\!\!- xx) \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr).(xx \in id\_reg\_8 \wedge xxr \in BYTE \Rightarrow rgs8 \lhd\!\!- \{\ xx \mapsto xxr\ \} \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr,yy,yyr).(xx \in id\_reg\_8 \wedge xxr \in BYTE \wedge yy \in id\_reg\_8 \wedge yyr \in BYTE \wedge \neg\ (xx{=}yy)$
$\Rightarrow rgs8 \lhd\!\!- \{\ xx \mapsto xxr, yy \mapsto yyr\ \} \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr,yy,yyr,zz,zzr).(xx \in id\_reg\_8 \wedge xxr \in BYTE \wedge yy \in id\_reg\_8 \wedge yyr \in BYTE \wedge zz \in id\_reg\_8 \wedge$
$zzr \in BYTE \wedge \neg\ (xx{=}yy) \wedge \neg\ (xx{=}zz) \wedge \neg\ (yy{=}zz)$
$\Rightarrow rgs8 \lhd\!\!- \{\ xx \mapsto xxr, yy \mapsto yyr\ ,\ zz \mapsto zzr\ \} \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr,yy,yyr,zz,zzr,vv,vvr).(xx \in id\_reg\_8 \wedge xxr \in BYTE \wedge yy \in id\_reg\_8 \wedge yyr \in BYTE \wedge zz \in id\_reg\_8 \wedge$
$zzr \in BYTE \wedge vv \in id\_reg\_8 \wedge vvr \in BYTE \wedge$
$\neg\ (xx{=}yy) \wedge \neg\ (xx{=}zz) \wedge \neg\ (xx{=}vv) \wedge \neg\ (yy{=}zz) \wedge \neg\ (yy{=}vv) \wedge \neg\ (zz{=}vv)$
$\Rightarrow rgs8 \lhd\!\!- \{\ xx \mapsto xxr, yy \mapsto yyr\ ,\ zz \mapsto zzr\ ,\ vv \mapsto vvr\ \} \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr,yy,yyr,zz,zzr,vv,vvr,ww,wwr).$
$(xx \in id\_reg\_8 \wedge xxr \in BYTE \wedge yy \in id\_reg\_8 \wedge yyr \in BYTE \wedge zz \in id\_reg\_8 \wedge zzr \in BYTE \wedge$
$vv \in id\_reg\_8 \wedge vvr \in BYTE \wedge ww \in id\_reg\_8 \wedge wwr \in BYTE \wedge \neg\ (xx{=}yy) \wedge \neg\ (xx{=}zz) \wedge$
$\neg\ (xx{=}vv) \wedge \neg\ (xx{=}ww) \wedge \neg\ (yy{=}zz) \wedge \neg\ (yy{=}vv) \wedge \neg\ (yy{=}ww) \wedge \neg\ (zz{=}vv) \wedge \neg\ (zz{=}ww) \wedge \neg\ (vv{=}ww)$
$\Rightarrow rgs8 \lhd\!\!- \{\ xx \mapsto xxr, yy \mapsto yyr\ ,\ zz \mapsto zzr\ ,\ vv \mapsto vvr,\ ww \mapsto wwr\ \} \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr,yy,yyr,zz,zzr,vv,vvr,ww,wwr,aa,aar).$
$(xx \in id\_reg\_8 \wedge xxr \in BYTE \wedge yy \in id\_reg\_8 \wedge yyr \in BYTE \wedge zz \in id\_reg\_8 \wedge zzr \in BYTE \wedge$
$vv \in id\_reg\_8 \wedge vvr \in BYTE \wedge ww \in id\_reg\_8 \wedge wwr \in BYTE \wedge aa \in id\_reg\_8 \wedge aar \in BYTE \wedge$
$\neg\ (xx{=}yy) \wedge \neg\ (xx{=}zz) \wedge \neg\ (xx{=}vv) \wedge \neg\ (xx{=}ww) \wedge \neg\ (xx{=}aa) \wedge \neg\ (yy{=}zz) \wedge \neg\ (yy{=}vv) \wedge \neg\ (yy{=}ww)$
$\wedge \neg\ (yy{=}aa) \wedge \neg\ (zz{=}vv) \wedge \neg\ (zz{=}ww) \wedge \neg\ (zz{=}aa) \wedge \neg\ (vv{=}ww) \wedge \neg\ (vv{=}aa) \wedge \neg\ (ww{=}aa)$
$\Rightarrow rgs8 \lhd\!\!- \{\ xx \mapsto xxr, yy \mapsto yyr\ ,\ zz \mapsto zzr\ ,\ vv \mapsto vvr,\ ww \mapsto wwr,\ aa \mapsto aar\ \} \in id\_reg\_8 \rightarrow BYTE) \wedge$

$\forall\ (xx,xxr,yy,yyr,zz,zzr,vv,vvr,ww,wwr,aa,aar,bb,bbr).$
$(xx \in id\_reg\_8 \wedge xxr \in BYTE \wedge yy \in id\_reg\_8 \wedge yyr \in BYTE \wedge zz \in id\_reg\_8 \wedge zzr \in BYTE \wedge$
$vv \in id\_reg\_8 \wedge vvr \in BYTE \wedge ww \in id\_reg\_8 \wedge wwr \in BYTE \wedge aa \in id\_reg\_8 \wedge aar \in BYTE \wedge$

$$
\begin{array}{l}
bb \in id\_reg\_8 \wedge bbr \in BYTE \wedge \neg (xx{=}yy) \wedge \neg (xx{=}zz) \wedge \neg (xx{=}vv) \wedge \neg (xx{=}ww) \wedge \\
\neg (xx{=}aa) \wedge \neg (xx{=}bb) \wedge \neg (yy{=}zz) \\
\wedge \neg (yy{=}vv) \wedge \neg (yy{=}ww) \quad \wedge \neg (yy{=}aa) \wedge \neg (yy{=}bb) \wedge \neg (zz{=}vv) \wedge \neg (zz{=}ww) \wedge \neg (zz{=}aa) \wedge \\
\neg (zz{=}bb) \wedge \neg (vv{=}ww) \wedge \neg (vv{=}aa) \wedge \neg (vv{=}bb) \wedge \neg (ww{=}aa) \wedge \neg (ww{=}bb) \wedge \neg (aa{=}bb) \\
\Rightarrow \quad rgs8 \Leftarrow \{\ xx \mapsto xxr, yy \mapsto yyr\ ,\ zz \mapsto zzr\ ,\quad vv \mapsto vvr,\quad ww \mapsto wwr,\ aa \mapsto aar,\ bb \mapsto bbr\ \} \\
\in id\_reg\_8 \rightarrow BYTE) \wedge
\end{array}
$$

$$
\begin{array}{l}
\mathbf{dom}(bv\_ireg\_plus\_d) = (\ BV16 \times SCHAR\ ) \wedge \\
\mathbf{ran}(bv\_ireg\_plus\_d) = (\ BV16\ ) \wedge \\
\forall\ (dd).(dd \in SCHAR \Rightarrow \quad bv\_ireg\_plus\_d(ix,dd) \in BV16) \wedge \\
\forall\ (dd).(dd \in SCHAR \Rightarrow \quad bv\_ireg\_plus\_d(iy,dd) \in BV16) \wedge
\end{array}
$$

$$
\begin{array}{l}
\mathbf{dom}(bv\_9ireg\_plus\_d0) = (\ (BV16 \rightarrow BYTE) \times BV16 \times SCHAR) \wedge \\
\mathbf{ran}(bv\_9ireg\_plus\_d0) = (\ BYTE\ ) \wedge
\end{array}
$$

$$
\begin{array}{l}
\forall\ (dd).(dd \in SCHAR \Rightarrow \quad bv\_9ireg\_plus\_d0(mem,ix,dd) \in BYTE) \wedge \\
\forall\ (dd).(dd \in SCHAR \Rightarrow \quad bv\_9ireg\_plus\_d0(mem,iy,dd) \in BYTE) \wedge
\end{array}
$$

$$
\begin{array}{l}
bv\_BC \quad \in BV16 \wedge \\
bv\_HL \in BV16 \wedge \\
bv\_DE \in BV16 \wedge \\
bv\_AF \in BV16 \wedge
\end{array}
$$

$$
\begin{array}{l}
bv\_9BC0 \in BYTE \wedge \\
bv\_9DE0 \in BYTE \wedge \\
bv\_9HL0 \in BYTE \wedge \\
bv\_9AF0 \in BYTE \wedge
\end{array}
$$

$$
\begin{array}{l}
bv\_9SP0 \in BYTE \wedge \\
bv\_9IX0 \in BYTE \wedge \\
bv\_9IY0 \in BYTE \wedge
\end{array}
$$

$$
\begin{array}{l}
dec\_BV16(bv\_BC) \quad \in BV16 \wedge \\
inc\_BV16(bv\_BC) \quad \in BV16 \wedge
\end{array}
$$

$$
\begin{array}{l}
dec\_BV16(bv\_DE) \quad \in BV16 \wedge \\
inc\_BV16(bv\_DE) \quad \in BV16 \wedge
\end{array}
$$

$$
\begin{array}{l}
inc\_BV16(bv\_HL) \quad \in BV16 \wedge \\
dec\_BV16(bv\_HL) \quad \in BV16 \wedge
\end{array}
$$

$$
\begin{array}{l}
sp\_plus\_one \in BV16 \wedge \\
sp\_plus\_two \in BV16 \wedge \\
sp\_minus\_one \in BV16 \wedge \\
sp\_minus\_two \in BV16 \wedge
\end{array}
$$

$$
\begin{array}{l}
z\_s \quad \in BIT \wedge \\
z\_z \quad \in BIT \wedge \\
z\_00 \quad \in BIT \wedge \\
z\_h \quad \in BIT \wedge \\
z\_01 \quad \in BIT \wedge \\
z\_p \quad \in BIT \wedge \\
z\_n \quad \in BIT \wedge \\
z\_c \quad \in BIT \wedge
\end{array}
$$

$io\_ports(rgs8(c0)) \in BYTE \wedge$

$dec(rgs8(b0)) \in BYTE$

INITIALISATION

$rgs8 := \{ (a0 \mapsto [1,1,1,1,1,1,1,1]) , (f0 \mapsto [1,1,1,1,1,1,1,1]) , (f\_0 \mapsto [1,1,1,1,1,1,1,1]) , (a\_0 \mapsto [1,1,1,1,1,1,1,1]) ,$
$(b0 \mapsto [1,1,1,1,1,1,1,1]) , (c0 \mapsto [1,1,1,1,1,1,1,1]) , (b\_0 \mapsto [1,1,1,1,1,1,1,1]) , (c\_0 \mapsto [1,1,1,1,1,1,1,1]) ,$
$(d0 \mapsto [1,1,1,1,1,1,1,1]) , (e0 \mapsto [1,1,1,1,1,1,1,1]) , (d\_0 \mapsto [1,1,1,1,1,1,1,1]) , (e\_0 \mapsto [1,1,1,1,1,1,1,1]) ,$
$(h0 \mapsto [1,1,1,1,1,1,1,1]) , (l0 \mapsto [1,1,1,1,1,1,1,1]) , (h\_0 \mapsto [1,1,1,1,1,1,1,1]) , (l\_0 \mapsto [1,1,1,1,1,1,1,1]) \} ||$
$pc := 0 || sp := [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] ||$
$ix := [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] || iy := [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1] ||$
$i\_ := uchar\_byte(0) || r\_ := [0,0,0,0,0,0,0,0] ||$
$io\_ports :\in (BYTE \rightarrow \{[0,0,0,0,0,0,0,0]\}) ||$
$iff1 := 0 || iff2 := 0 ||$
$im := (0 \mapsto 0)$

OPERATIONS

ext_update_io_ports($address$,$value$)=
PRE $address \in UCHAR \wedge value \in SCHAR$ **THEN**
io_ports ($uchar\_byte$ ($address$)) := $schar\_byte$ ($value$)
END;

IN_A_9n0 (nn) =
PRE $nn \in UCHAR$ **THEN**
rgs8 ($a0$) := io_ports ($uchar\_byte$ (nn)) ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

IN_r_9C0 (rr) =
PRE $rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**
ANY
$negative$, $zero$, $half\_carry$, pv , add_sub , $carry$
WHERE
$negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$negative = is\_negative$ (io_ports ($rgs8$ ($c0$))) $\wedge$
$zero = is\_zero$ (io_ports ($rgs8$ ($c0$))) $\wedge$
$half\_carry = 0 \wedge$
$pv = parity\_even$ (io_ports ($rgs8$ ($c0$))) $\wedge$
$add\_sub = 0 \wedge$
$carry = z\_c$
THEN
$rgs8 := rgs8 \Leftarrow \{$ ($rr \mapsto io\_ports$ ($rgs8$ ($c0$))) ,
$update\_flag\_reg$($negative$, $zero$, $half\_carry$, pv , add_sub , $carry$) $\} ||$
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END
END;

INI =
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
ANY hvn , lvn ,
$negative$, $zero$, $half\_carry$, pv , add_sub , $carry$
WHERE

$negative \in BIT \wedge \quad zero \in BIT \wedge \quad half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge$
$hvn , lvn = bv16\_byte (inc\_BV16 (bv\_HL)) \wedge$
$negative = is\_negative (io\_ports (rgs8 (c0))) \wedge$
$zero = is\_zero (dec (rgs8 (b0))) \wedge$
$half\_carry = z\_h \wedge$
$pv = parity\_even (io\_ports (rgs8 (c0))) \quad \wedge$
$add\_sub = 1 \wedge$
$carry = z\_c$

THEN

updateAddressMem ($bv\_HL , io\_ports (rgs8 (c0))) \parallel$
$rgs8 := rgs8 \Leftarrow \{ (h0 \mapsto hvn) , (l0 \mapsto lvn) , (b0 \mapsto dec (rgs8 (b0))) ,$
$update\_flag\_reg(negative , zero , half\_carry , pv , add\_sub , carry) \} \parallel$
$pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$

END

END;

INIR =
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**

ANY $hvn , lvn ,$
$negative , zero , half\_carry , pv , add\_sub , carry$

WHERE

$negative \in BIT \wedge \quad zero \in BIT \wedge \quad half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge$
$hvn , lvn = bv16\_byte (inc\_BV16 (bv\_HL)) \wedge$
$negative = is\_negative (io\_ports (rgs8 (c0))) \wedge$
$zero = 1 \quad \wedge$
$half\_carry = z\_h \wedge$
$pv = parity\_even (io\_ports (rgs8 (c0))) \quad \wedge$
$add\_sub = 1 \wedge$
$carry = z\_c$

THEN

updateAddressMem ($bv\_HL , io\_ports (rgs8 (c0))) \parallel$
$rgs8 := rgs8 \Leftarrow \{ (h0 \mapsto hvn) , (l0 \mapsto lvn) , (b0 \mapsto dec (rgs8 (b0))) ,$
$update\_flag\_reg(negative , zero , half\_carry , pv , add\_sub , carry) \} \parallel$
$r\_ := update\_refresh\_reg(r\_) \parallel$
IF $is\_zero (dec (rgs8 (b0))) = 1$ **THEN** $pc := instruction\_next (pc)$ **END**

END

END;

IND =
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**

ANY $hvn , lvn ,$
$negative , zero , half\_carry , pv , add\_sub , carry$

WHERE

$negative \in BIT \wedge \quad zero \in BIT \wedge \quad half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge$
$hvn , lvn = bv16\_byte (dec\_BV16 (bv\_HL)) \wedge$
$negative = is\_negative (io\_ports (rgs8 (c0))) \wedge$
$zero = is\_zero (dec (rgs8 (b0))) \wedge$
$half\_carry = z\_h \wedge$
$pv = parity\_even (io\_ports (rgs8 (c0))) \quad \wedge$
$add\_sub = 1 \wedge$
$carry = z\_c$

```
    THEN
        updateAddressMem ( bv_HL , io_ports ( rgs8 ( c0 ) ) ) ||
        rgs8 :=  rgs8 <+ { ( h0 ↦ hvn ) , ( l0 ↦ lvn ) , ( b0 ↦ dec ( rgs8 ( b0 ) ) ) ,
            update_flag_reg( negative , zero , half_carry , pv , add_sub , carry ) } ||
        pc := instruction_next ( pc ) || r_ := update_refresh_reg(r_)
    END
END;

INDR =
PRE bv16_ushort(bv_HL) ∈ DATA_R_ADR    THEN
    ANY  hvn , lvn ,
        negative , zero , half_carry , pv , add_sub , carry
    WHERE
        negative ∈ BIT ∧   zero ∈ BIT ∧   half_carry ∈ BIT ∧ pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
        hvn ∈ BYTE ∧ lvn ∈ BYTE ∧
        hvn , lvn = bv16_byte ( dec_BV16 ( bv_HL ) ) ∧
        negative = is_negative ( io_ports ( rgs8 ( c0 ) ) ) ∧
        zero = is_zero ( dec ( rgs8 ( b0 ) ) ) ∧
        half_carry = z_h ∧
        pv = parity_even ( io_ports ( rgs8 ( c0 ) ) )    ∧
        add_sub =  1 ∧
        carry = z_c
    THEN
        updateAddressMem ( bv_HL , io_ports ( rgs8 ( c0 ) ) ) ||
        rgs8 :=  rgs8 <+ { ( h0 ↦ hvn ) , ( l0 ↦ lvn ) , ( b0 ↦ dec ( rgs8 ( b0 ) ) ) ,
            update_flag_reg( negative , zero , half_carry , pv , add_sub , carry ) } ||
        IF  is_zero ( dec ( rgs8 ( b0 ) ) ) = 1
        THEN pc := instruction_next ( pc ) || r_ := update_refresh_reg(r_) END
    END
END;

OUT_9n0_A ( nn ) =
PRE nn ∈ UCHAR    THEN
    io_ports ( uchar_byte ( nn ) ) := rgs8 ( a0 ) ||
    pc := instruction_next ( pc ) || r_ := update_refresh_reg(r_)
END;

OUT_9C0_r ( rr ) =
PRE rr ∈ id_reg_8 ∧ rr ≠ f0  THEN
    io_ports ( rgs8 ( rr ) ) := rgs8 ( c0 ) ||
    pc := instruction_next ( pc ) || r_ := update_refresh_reg(r_)
END;

OUTI =
ANY hvn , lvn ,
    negative , zero , half_carry , pv , add_sub , carry
WHERE
    negative ∈ BIT ∧   zero ∈ BIT ∧   half_carry ∈ BIT ∧ pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
    hvn ∈ BYTE ∧ lvn ∈ BYTE ∧
    hvn , lvn = bv16_byte ( inc_BV16 ( bv_HL ) ) ∧
    negative = is_negative ( bv_9HL0 ) ∧
    zero = is_zero ( dec ( rgs8 ( b0 ) ) ) ∧
    half_carry = z_h ∧
    pv = parity_even ( bv_9HL0 ) ∧
```

$add\_sub = 1 \wedge$
$carry = z\_c$
THEN
io_ports ($rgs8$ ($c0$)) := bv_9HL0 ||
updateAddressMem (bv_HL , io_ports ($rgs8$ ($c0$))) ||
$rgs8 := rgs8 \Leftarrow \{ (h0 \mapsto hvn) , (l0 \mapsto lvn) , (b0 \mapsto dec (rgs8 (b0))) ,$
$update\_flag\_reg(negative , zero , half\_carry , pv , add\_sub , carry) \}$ ||
$pc := instruction\_next (pc)$ || $r\_ := update\_refresh\_reg(r\_)$

END;

OUTIR =
ANY $hvn , lvn ,$
$negative , zero , half\_carry , pv , add\_sub , carry$
WHERE
$negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge$
$hvn , lvn = bv16\_byte (inc\_BV16 (bv\_HL)) \wedge$
$negative = is\_negative (bv\_9HL0) \wedge$
$zero = is\_zero (dec (rgs8 (b0))) \wedge$
$half\_carry = z\_h \wedge$
$pv = parity\_even (bv\_9HL0) \wedge$
$add\_sub = 1 \wedge$
$carry = z\_c$
THEN
io_ports ($rgs8$ ($c0$)) := bv_9HL0 ||
updateAddressMem (bv_HL , io_ports ($rgs8$ ($c0$))) ||
$rgs8 := rgs8 \Leftarrow \{ (h0 \mapsto hvn) , (l0 \mapsto lvn) , (b0 \mapsto dec (rgs8 (b0))) ,$
$update\_flag\_reg(negative , zero , half\_carry , pv , add\_sub , carry) \}$ ||
$r\_ := update\_refresh\_reg(r\_)$ ||
IF $is\_zero (dec (rgs8 (b0))) = 1$ **THEN** $pc := instruction\_next (pc)$ **END**
END;

OUTD =
ANY $hvn , lvn ,$
$negative , zero , half\_carry , pv , add\_sub , carry$
WHERE
$negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge$
$hvn , lvn = bv16\_byte (dec\_BV16 (bv\_HL)) \wedge$
$negative = is\_negative (bv\_9HL0) \wedge$
$zero = is\_zero (dec (rgs8 (b0))) \wedge$
$half\_carry = z\_h \wedge$
$pv = parity\_even (bv\_9HL0) \wedge$
$add\_sub = 1 \wedge$
$carry = z\_c$
THEN
io_ports ($rgs8$ ($c0$)) := bv_9HL0 ||
updateAddressMem (bv_HL , io_ports ($rgs8$ ($c0$))) ||
$rgs8 := rgs8 \Leftarrow \{ (h0 \mapsto hvn) , (l0 \mapsto lvn) , (b0 \mapsto dec (rgs8 (b0))) ,$
$update\_flag\_reg(negative , zero , half\_carry , pv , add\_sub , carry) \}$ ||
$pc := instruction\_next (pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

OUTDR =
ANY hvn , lvn ,
$negative$, $zero$, $half\_carry$, pv , add_sub , $carry$
WHERE
$negative \in BIT \wedge \quad zero \in BIT \wedge \quad half\_carry \in BIT \wedge pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge$
hvn , $lvn = bv16\_byte\ (\ dec\_BV16\ (\ bv\_HL\ )\ ) \wedge$
$negative = is\_negative\ (\ bv\_9HL0\ ) \wedge$
$zero = is\_zero\ (\ dec\ (\ rgs8\ (\ b0\ )\ )\ ) \wedge$
$half\_carry = z\_h \wedge$
$pv = parity\_even\ (\ bv\_9HL0\ ) \wedge$
$add\_sub = \ 1 \wedge$
$carry = z\_c$
THEN
io_ports $(\ rgs8\ (\ c0\ )\ ) := bv\_9HL0$ ||
updateAddressMem $(\ bv\_HL$, $io\_ports\ (\ rgs8\ (\ c0\ )\ )\ )$ ||
$rgs8 := \ \ rgs8 \Leftarrow \{\ (\ h0 \mapsto hvn\ )$, $(\ l0 \mapsto lvn\ )$, $(\ b0 \mapsto dec\ (\ rgs8\ (\ b0\ )\ )\ )$,
$update\_flag\_reg(\ negative$, $zero$, $half\_carry$, pv , add_sub , $carry\ )\ \}$ ||
$r\_ := update\_refresh\_reg(r\_)$ ||
IF $\ is\_zero\ (\ dec\ (\ rgs8\ (\ b0\ )\ )\ ) = 1$ **THEN** $pc := instruction\_next\ (\ pc\ )$ **END**
END

RLCA =
BEGIN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto rotateleft(rgs8(a0))$,
$update\_flag\_reg(z\_s,z\_z,0,z\_p,0,bitget(rgs8(a0),7)\ )\quad \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RLA =
BEGIN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto (rotateleft(rgs8(a0)) \Leftarrow \{(1 \mapsto z\_c)\}\ )$,
$update\_flag\_reg(z\_s,z\_z,0,z\_p,0,bitget(rgs8(a0),7)\ )\quad \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RRCA =
BEGIN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto rotateright(rgs8(a0))$,
$update\_flag\_reg(z\_s,z\_z,0,z\_p,0,bitget(rgs8(a0),0)\ )\quad \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RRA =
BEGIN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto (rotateright(rgs8(a0)) \Leftarrow \{(8 \mapsto z\_c)\}\ )$,
$update\_flag\_reg(z\_s,z\_z,0,z\_p,0,bitget(rgs8(a0),0)\ )\quad \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RLC_r(rr) =
PRE $rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateleft(rgs8(rr))$ **THEN**
$rgs8 := rgs8 \Leftarrow \{\ rr \mapsto res$,

$update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(rgs8(rr),7)\ )\quad \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

RLC_9HL0 =
PRE $bv16\_ushort(\ bv\_HL\ )\quad \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateleft(\ bv\_9HL0\ )$ **THEN**
$rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg(\quad is\_negative(res),\ is\_zero(res),0,$
$parity\_even(res)\ ,0,bitget(bv\_9HL0,7)\ )\quad \}\ ||$
updateAddressMem$(bv\_HL,\ res\ )\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

RLC_9IX_d0$(desloc)$ =
PRE $desloc \in SCHAR \wedge bv16\_ushort\ (\ bv\_ireg\_plus\_d(ix,desloc)) \in \quad DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateleft(\ bv\_9ireg\_plus\_d0(mem,ix,desloc)\ )$ **THEN**
$rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,ix,desloc),7)\ )\quad \}\ ||$
updateAddressMem$(bv\_ireg\_plus\_d(ix,desloc),\ res\ )\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

RLC_9IY_d0$(desloc)$ =
PRE $desloc \in SCHAR \wedge bv16\_ushort\ (bv\_ireg\_plus\_d(iy,desloc)) \in \quad DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateleft(\ bv\_9ireg\_plus\_d0(mem,iy,desloc)\ )$ **THEN**
$rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),7)\ )\quad \}\ ||$
updateAddressMem$(bv\_ireg\_plus\_d(iy,desloc),\ res\ )\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

RL_r(rr) =
PRE $rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = (rotateleft(rgs8(rr)) \Leftarrow \{(1 \mapsto z\_c)\}\ )$ **THEN**
$rgs8 := rgs8 \Leftarrow \{\ rr \mapsto res,$
$update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(rgs8(rr),7)\ )\quad \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

RL_9HL0 =
PRE $bv16\_ushort\ (bv\_HL) \in \quad DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = (rotateleft(\ bv\_9HL0\ ) \Leftarrow \{(\ 0 \mapsto z\_c)\}\ )$ **THEN**
$rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(bv\_9HL0,7)\ )\quad \}\ ||$
updateAddressMem$(bv\_HL,\ res\ )\ ||$

$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END ;

RL_9IX_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = (rotateleft(\ bv\_9ireg\_plus\_d0(mem,ix,desloc)\ ) \lhd\!\!\!- \{(1 \mapsto z\_c)\}\ )$
THEN
$rgs8 := rgs8 \lhd\!\!\!- \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res),$
$0,\ bitget(bv\_9ireg\_plus\_d0(mem,ix,desloc),7)\ )\ \ \}$ ||
updateAddressMem($bv\_ireg\_plus\_d(ix,desloc),\ res\ )$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

RL_9IY_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = (rotateleft(\ bv\_9ireg\_plus\_d0(mem,iy,desloc)\ ) \lhd\!\!\!- \{(1 \mapsto z\_c)\}\ )$
THEN
$rgs8 := rgs8 \lhd\!\!\!- \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),7)\ )\ \ \}$ ||
updateAddressMem($bv\_ireg\_plus\_d(iy,desloc),\ res\ )$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

RRC_r(rr) =
PRE $rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateright(rgs8(rr))$ **THEN**
$rgs8 := rgs8 \lhd\!\!\!- \{\ rr \mapsto res,$
$update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(rgs8(rr),0)\ )\ \ \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

RRC_9HL0 =
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateright(\ bv\_9HL0\ )$ **THEN**
$rgs8 := rgs8 \lhd\!\!\!- \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(bv\_9HL0,0)\ )\ \ \}$ ||
updateAddressMem($bv\_HL,\ res\ )$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

RRC_9IX_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = rotateright(\ bv\_9ireg\_plus\_d0(mem,ix,desloc)\ )$ **THEN**
$rgs8 := rgs8 \lhd\!\!\!- \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,ix,desloc),0)\ )\ \ \}$ ||

```
        updateAddressMem(bv_ireg_plus_d(ix,desloc), res ) ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

RRC_9IY_d0(desloc) =
PRE desloc ∈ SCHAR ∧ bv16_ushort(bv_ireg_plus_d(iy,desloc)) ∈   DATA_R_ADR  THEN
    ANY res WHERE res ∈ BYTE ∧ res = rotateright( bv_9ireg_plus_d0(mem,iy,desloc) ) THEN
        rgs8 := rgs8 <+ { update_flag_reg(
                is_negative(res), is_zero(res),0, parity_even(res),
                0,bitget(bv_9ireg_plus_d0(mem,iy,desloc),0) )    } ||
        updateAddressMem(bv_ireg_plus_d(iy,desloc), res ) ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

RR_r(rr) =
PRE rr ∈ id_reg_8 ∧ rr ≠ f0 THEN
    ANY res WHERE res ∈ BYTE ∧ res = (rotateright(rgs8(rr)) <+ {(1 ↦ z_c)} ) THEN
        rgs8 := rgs8 <+ { rr ↦ res,
            update_flag_reg(
                is_negative(res), is_zero(res),0, parity_even(res) ,0,bitget(rgs8(rr),0) )    } ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

RR_9HL =
PRE bv16_ushort(bv_HL) ∈ STACK_R_ADR THEN
    ANY res WHERE res ∈ BYTE ∧ res = (rotateright( bv_9HL0 ) <+ {(1 ↦ z_c)} ) THEN
        rgs8 := rgs8 <+ { update_flag_reg(
                is_negative(res), is_zero(res),0, parity_even(res) ,0,bitget(bv_9HL0,0) )    } ||
        updateAddressMem(bv_HL, res ) ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

RR_9IX_d0(desloc) =
PRE desloc ∈ SCHAR ∧ bv16_ushort(bv_ireg_plus_d(ix,desloc)) ∈ DATA_R_ADR THEN
    ANY res WHERE res ∈ BYTE ∧ res = (rotateright( bv_9ireg_plus_d0(mem,ix,desloc) ) <- {(1 ↦ z_c)} )
    THEN
        rgs8 := rgs8 <+ { update_flag_reg(
                is_negative(res), is_zero(res),0, parity_even(res),
                0,bitget(bv_9ireg_plus_d0(mem,ix,desloc),0) )    } ||
        updateAddressMem(bv_ireg_plus_d(ix,desloc), res ) ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

RR_9IY_d0(desloc) =
PRE desloc ∈ SCHAR ∧ bv16_ushort(bv_ireg_plus_d(iy,desloc)) ∈ DATA_R_ADR THEN
    ANY res WHERE res ∈ BYTE ∧ res = (rotateright( bv_9ireg_plus_d0(mem,iy,desloc) ) <- {(1 ↦ z_c)}
)
    THEN
        rgs8 := rgs8 <+ { update_flag_reg(
```

$$
\begin{array}{l}
\quad\quad\quad\quad is\_negative(res),\ is\_zero(res),0,\ parity\_even(res), \\
\quad\quad\quad\quad 0,bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),0)\ )\quad \}\ || \\
\quad\quad \mathbf{updateAddressMem}(bv\_ireg\_plus\_d(iy,desloc),\ res\ )\ || \\
\quad\quad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_) \\
\quad \mathbf{END} \\
\mathbf{END};
\end{array}
$$

$$
\begin{array}{l}
\mathbf{SLA\_r}(rr) = \\
\mathbf{PRE}\ rr \in id\_reg\_8 \wedge rr \neq f0\ \mathbf{THEN} \\
\quad \mathbf{ANY}\ res\ \mathbf{WHERE}\ res \in BYTE \wedge res = (rotateright(rgs8(rr)) \Leftarrow \{(1 \mapsto z\_c)\}\ )\ \mathbf{THEN} \\
\quad\quad rgs8 := rgs8 \Leftarrow \{\ rr \mapsto res, \\
\quad\quad\quad update\_flag\_reg(\\
\quad\quad\quad\quad is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(rgs8(rr),7)\ )\quad \}\ || \\
\quad\quad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_) \\
\quad \mathbf{END} \\
\mathbf{END};
\end{array}
$$

$$
\begin{array}{l}
\mathbf{SLA\_9HL0} = \\
\mathbf{PRE}\ \ bv16\_ushort(bv\_HL) \in DATA\_R\_ADR\ \mathbf{THEN} \\
\quad \mathbf{ANY}\ res\ \mathbf{WHERE}\ res \in BYTE \wedge res = (rotateright(\ bv\_9HL0\ ) \Leftarrow \{(1 \mapsto z\_c)\}\ )\ \mathbf{THEN} \\
\quad\quad rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg(\\
\quad\quad\quad\quad is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(bv\_9HL0,7)\ )\quad \}\ || \\
\quad\quad \mathbf{updateAddressMem}(bv\_HL,\ res\ )\ || \\
\quad\quad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_) \\
\quad \mathbf{END} \\
\mathbf{END};
\end{array}
$$

$$
\begin{array}{l}
\mathbf{SRA\_9HL0} = \\
\mathbf{PRE}\ \ bv16\_ushort(bv\_HL) \in DATA\_R\_ADR\ \ \mathbf{THEN} \\
\quad \mathbf{ANY}\ res\ \mathbf{WHERE}\ res \in BYTE \wedge res = (rotateright(\ bv\_9HL0\ )) \Leftarrow \{8 \mapsto bv\_9HL0(7)\}\ \ \mathbf{THEN} \\
\quad\quad rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg(\\
\quad\quad\quad\quad is\_negative(res),\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(bv\_9HL0,0)\ )\quad \}\ || \\
\quad\quad \mathbf{updateAddressMem}(bv\_HL,\ res\ )\ || \\
\quad\quad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_) \\
\quad \mathbf{END} \\
\mathbf{END};
\end{array}
$$

$$
\begin{array}{l}
\mathbf{SRA\_9IX\_d0}(desloc) = \\
\mathbf{PRE}\ desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR\ \ \mathbf{THEN} \\
\quad \mathbf{ANY}\ res\ \mathbf{WHERE}\ res \in BYTE \wedge \\
\quad\quad res = (rotateright(\ bv\_9ireg\_plus\_d0(mem,ix,desloc)\ )) \Leftarrow \\
\quad\quad \{7 \mapsto bitget(bv\_9ireg\_plus\_d0(mem,ix,desloc),7)\ \} \\
\quad \mathbf{THEN} \\
\quad\quad rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg(\\
\quad\quad\quad\quad is\_negative(res),\ is\_zero(res),0,\ parity\_even(res), \\
\quad\quad\quad\quad 0,bitget(bv\_9ireg\_plus\_d0(mem,ix,desloc),0)\ )\quad \}\ || \\
\quad\quad \mathbf{updateAddressMem}(bv\_ireg\_plus\_d(ix,desloc),\ res\ )\ || \\
\quad\quad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_) \\
\quad \mathbf{END} \\
\mathbf{END};
\end{array}
$$

$$
\begin{array}{l}
\mathbf{SRA\_9IY\_d0}(desloc) = \\
\mathbf{PRE}\ desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR\ \ \mathbf{THEN} \\
\quad \mathbf{ANY}\ res\ \mathbf{WHERE}\ res \in BYTE \wedge
\end{array}
$$

$res = rotateright(\ bv\_9ireg\_plus\_d0(mem,iy,desloc)\ ) \leftarrow$
$\{7 \mapsto bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),7)\ \}$
THEN
$rgs8 := rgs8 \leftarrow \{\ update\_flag\_reg($
$is\_negative(res),\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),0)\ )\quad \}\ ||$
updateAddressMem$(bv\_ireg\_plus\_d(ix,desloc),\ res\ )\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

SRL_r$(rr) =$
PRE $rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = bitclear(rotateright(rgs8(rr)),7)$ **THEN**
$rgs8 := rgs8 \leftarrow \{\ rr \mapsto res,$
$update\_flag\_reg($
$0,\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(rgs8(rr),0)\ )\quad \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

SRL_9HL0 $=$
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge res = bitclear(rotateright(\ bv\_9HL0\ ),7)$ **THEN**
$rgs8 := rgs8 \leftarrow \{\ update\_flag\_reg($
$0,\ is\_zero(res),0,\ parity\_even(res)\ ,0,bitget(bv\_9HL0,0)\ )\quad \}\ ||$
updateAddressMem$(bv\_HL,\ res\ )\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

SRL_9IX_d0$(desloc) =$
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge$
$res = bitclear(rotateright(\ bv\_9ireg\_plus\_d0(mem,ix,desloc)\ ),7)$
THEN
$rgs8 := rgs8 \leftarrow \{\ update\_flag\_reg($
$0,\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,ix,desloc),0)\ )\quad \}\ ||$
updateAddressMem$(bv\_ireg\_plus\_d(ix,desloc),\ res\ )\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

SRL_9IY_d0$(desloc) =$
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY res **WHERE** $res \in BYTE \wedge$
$res = bitclear(rotateright(\ bv\_9ireg\_plus\_d0(mem,iy,desloc)\ ),7)$
THEN
$rgs8 := rgs8 \leftarrow \{\ update\_flag\_reg($
$0,\ is\_zero(res),0,\ parity\_even(res),$
$0,bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),0)\ )\quad \}\ ||$
updateAddressMem$(bv\_ireg\_plus\_d(iy,desloc),\ res\ )\quad ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$

```
    END
END;

RLD=
PRE bv16_ushort(bv_HL) ∈ DATA_R_ADR THEN
    ANY res, acc WHERE res ∈ BYTE ∧ acc ∈ BYTE ∧
        res =  { 8 ↦ bitget(bv_9HL0,3), 7 ↦ bitget(bv_9HL0,2), 6 ↦ bitget(bv_9HL0,1),
            5 ↦ bitget(bv_9HL0,0),
            4 ↦ bitget(rgs8(a0),3),   3 ↦ bitget(rgs8(a0),2), 2 ↦ bitget(rgs8(a0),1),
            1 ↦ bitget(rgs8(a0),0)    } ∧
        acc = rgs8(a0) <+ { 4 ↦ bitget(bv_9HL0,7),   3 ↦ bitget(bv_9HL0,6),
            2 ↦ bitget(bv_9HL0,5),    1 ↦ bitget(bv_9HL0,4)  }
    THEN
        rgs8 := rgs8 <+ { update_flag_reg(
                is_negative(acc), is_zero(acc),0, parity_even(acc),
                0,z_c )    } ||
        updateAddressMem(bv_HL, res )
        || pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

RRD=
PRE bv16_ushort(bv_HL) ∈ DATA_R_ADR THEN
    ANY res, acc WHERE res ∈ BYTE ∧ acc ∈ BYTE ∧
        res =  { 8 ↦ bitget(rgs8(a0),3),   7 ↦ bitget(rgs8(a0),2), 6 ↦ bitget(rgs8(a0),1),
            5 ↦ bitget(rgs8(a0),0),
            4 ↦ bitget(bv_9HL0,7), 3 ↦ bitget(bv_9HL0,6), 2 ↦ bitget(bv_9HL0,5),
            1 ↦ bitget(bv_9HL0,4)} ∧
        acc = rgs8(a0) <+ { 4 ↦ bitget(bv_9HL0,3),   3 ↦ bitget(bv_9HL0,2),
            2 ↦ bitget(bv_9HL0,1),    1 ↦ bitget(bv_9HL0,0)  }
    THEN
        rgs8 := rgs8 <+ { update_flag_reg( is_negative(acc), is_zero(acc),0, parity_even(acc), 0 ,z_c )    } ||
        updateAddressMem(bv_HL,res)
        || pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

BIT_b_rr (bb,rr) =
PRE bb ∈ 0 .. 7 ∧ rr ∈ id_reg_8 ∧ rr ≠ f0 THEN
    rgs8 := rgs8 <+ {update_flag_reg( z_s, bit_not( bitget( rgs8(rr),bb) ), 1,z_p,0,z_c)} ||
    pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
END;

BIT_b_9HL0 (bb) =
PRE bb ∈ 0 .. 7 THEN
    rgs8 := rgs8 <+ {update_flag_reg( z_s, bit_not( bitget(bv_9HL0,bb) ), 1,z_p,0,z_c)} ||
    pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
END;

BIT_b_9IX_d0 (desloc, bb) =
PRE bb ∈ 0 .. 7 ∧ desloc ∈ SCHAR THEN
    rgs8 := rgs8 <+ {update_flag_reg( z_s, bit_not( bitget(bv_9ireg_plus_d0(mem,ix,desloc),bb) ), 1,z_p,0,z_c)} ||
    pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
END;
```

BIT_b_9IY_d0 $(desloc,\ bb) =$
PRE $bb \in 0\ldots7 \wedge desloc \in SCHAR$ **THEN**
$rgs8 := rgs8 \Leftarrow \{update\_flag\_reg(\ z\_s,\ bit\_not(\ bitget(bv\_9ireg\_plus\_d0(mem,iy,desloc),bb)\ ),\ 1,z\_p,0,z\_c)\}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

SET_b_r$(bb,rr)=$
PRE $bb \in 0\ldots7 \wedge rr \in id\_reg\_8$ **THEN**
rgs8$(rr):= bitset(rgs8(rr),bb)$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

SET_b_9HL0$(bb)=$
PRE $bb \in 0\ldots7 \wedge bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
updateAddressMem$(bv\_HL,\ bitset(bv\_9HL0,bb))$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

SET_b_9IX_d0$(bb,desloc)=$
PRE $bb \in 0\ldots7 \wedge desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR$ **THEN**
updateAddressMem$(bv\_ireg\_plus\_d(ix,desloc),\ bitset(bv\_9ireg\_plus\_d0(mem,ix,desloc),bb))$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

SET_b_9IY_d0$(bb,desloc)=$
PRE $bb \in 0\ldots7 \wedge desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR$ **THEN**
updateAddressMem$(bv\_ireg\_plus\_d(iy,desloc),\ bitset(bv\_9ireg\_plus\_d0(mem,iy,desloc),bb))$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RES_b_r$(bb,rr)=$
PRE $bb \in 0\ldots7 \wedge rr \in id\_reg\_8 \wedge rr \neq f0$ **THEN**
rgs8$(rr):= bitclear(rgs8(rr),bb)$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RES_b_9HL0$(bb)=$
PRE $bb \in 0\ldots7 \wedge bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
updateAddressMem$(bv\_HL,\ bitclear(bv\_9HL0,bb))$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RES_b_9IX_d0$(bb,desloc)=$
PRE $bb \in 0\ldots7 \wedge desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR$ **THEN**
updateAddressMem$(bv\_ireg\_plus\_d(ix,desloc),\ bitclear(bv\_9ireg\_plus\_d0(mem,ix,desloc),bb))$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

RES_b_9IY_d0$(bb,desloc)=$
PRE $bb \in 0\ldots7 \wedge desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR$ **THEN**
updateAddressMem$(bv\_ireg\_plus\_d(iy,desloc),\ bitclear(bv\_9ireg\_plus\_d0(mem,iy,desloc),bb))$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

JP_nn(*nn*)=
PRE *nn* ∈ *USHORT* **THEN**
  pc:= *nn* || *r_* := *update_refresh_reg*(*r_*)
END;

JP_cc_nn(*cc*,*nn*)=
PRE *cc* ∈ 0 .. 7 ∧ *nn* ∈ *USHORT* **THEN**
  ANY *res* **WHERE** *res* ∈ *BIT* ∧
    { 0 ↦ *bit_not*(*z_z*) , 1 ↦ *z_z*,
      2 ↦ *bit_not*(*z_c*) , 3 ↦ *z_c* ,
      4 ↦ *bit_not*(*z_p*) , 5 ↦ *z_p* ,
      6 ↦ *bit_not*(*z_s*) , 7 ↦ *z_s* }(*cc*) = *res*
  THEN
    IF *res* = 1 **THEN** *pc*:= *nn*
    ELSE *pc* := *instruction_next*(*pc*) **END**
    || *r_* := *update_refresh_reg*(*r_*)
  END
END;

JR_e(*ee_p*)=
PRE *ee_p* ∈ *USHORT* ∧ *ee_p* - *pc* ≤ 129 ∧ *ee_p* - *pc* ≥ (-126) **THEN**
  pc := *ee_p* || *r_* := *update_refresh_reg*(*r_*)
END;

JR_C_e(*ee_p*)=
PRE *ee_p* ∈ *USHORT* ∧ *ee_p* - *pc* ≤ 129 ∧ *ee_p* - *pc* ≥ (-126) **THEN**
  IF *z_c* = 0 **THEN** *pc* := *instruction_next*(*pc*)
  ELSE *pc* := *ee_p* **END**
  || *r_* := *update_refresh_reg*(*r_*)
END;

JR_NC_e(*ee_p*)=
PRE *ee_p* ∈ *USHORT* ∧ *ee_p* - *pc* ≤ 129 ∧ *ee_p* - *pc* ≥ (-126) **THEN**
  IF *z_c* = 1 **THEN** *pc* := *instruction_next*(*pc*)
  ELSE *pc* := *ee_p* **END**
  || *r_* := *update_refresh_reg*(*r_*)
END;

JR_Z_e(*ee_p*)=
PRE *ee_p* ∈ *USHORT* ∧ *ee_p* - *pc* ≤ 129 ∧ *ee_p* - *pc* ≥ (-126) **THEN**
  IF *z_z* = 0 **THEN** *pc* := *instruction_next*(*pc*)
  ELSE *pc* := *ee_p* **END**
  || *r_* := *update_refresh_reg*(*r_*)
END;

JR_NZ_e(*ee_p*)=
PRE *ee_p* ∈ *USHORT* ∧ *ee_p* - *pc* ≤ 129 ∧ *ee_p* - *pc* ≥ (-126) **THEN**
  IF *z_z* = 1 **THEN** *pc* := *instruction_next*(*pc*)
  ELSE *pc* := *ee_p* **END**
  || *r_* := *update_refresh_reg*(*r_*)
END;

JP_HL=
BEGIN
  $pc := bv16\_ushort(bv\_HL) \;||\; r\_ := update\_refresh\_reg(r\_)$
END;

JP_IX=
BEGIN
  $pc := bv16\_ushort(\; ix) \;\;||\; r\_ := update\_refresh\_reg(r\_)$
END;

JP_IY=
BEGIN
  $pc := \;\; bv16\_ushort(\; iy) \;||\; r\_ := update\_refresh\_reg(r\_)$
END;

DJNZ_e(ee_p)=
PRE $ee\_p \in USHORT \wedge ee\_p$ - $pc \;\; \leq 129 \;\; \wedge ee\_p$ - $pc \;\; \geq$ (-126) **THEN**
  rgs8($b0$):= $\;\; dec(rgs8(b0)) \;||$
  IF $is\_zero(dec(rgs8(b0))) = 1$ **THEN** $\;\; pc := instruction\_next(pc)$
  ELSE $pc := ee\_p$ **END** $\;||\; r\_ := update\_refresh\_reg(r\_)$
END;

CALL_nn (nn) =
PRE $nn \in USHORT \wedge bv16\_ushort(sp\_minus\_two) \in STACK\_R\_ADR \;\; \wedge \;\; bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
  ANY $high,low$ **WHERE** $high \in BYTE \wedge low \in BYTE \wedge (\; high \; , \; low \;) = bv16\_byte(ushort\_bv16(pc)) \wedge$
    $sp\_minus\_one \neq sp\_minus\_two$
  THEN
    updateStack({ (sp_minus_one) $\mapsto$ $high$, (sp_minus_two) $\mapsto$ low }) $||$
    $sp := sp\_minus\_two \;||$
    $pc := nn$
  END
END;

CALL_cc_nn (cc,nn) =
PRE $cc \in 0 \,..\, 8 \wedge nn \in USHORT \wedge bv16\_ushort(sp\_minus\_two) \in STACK\_R\_ADR \;\; \wedge$
    $bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
  ANY $high,low$ **WHERE** $high \in BYTE \wedge low \in BYTE \wedge \;\; (\; high \; , \; low \;) = bv16\_byte(ushort\_bv16(pc))$
  THEN
    IF $cc\_get(rgs8,cc) = 1$ **THEN**
      updateStack({ (sp_minus_one) $\mapsto$ $high$, (sp_minus_two) $\mapsto$ low }) $||$
      $sp := sp\_minus\_two \;||$
      $pc := nn$
    ELSE
      $pc := instruction\_next(pc)$
    END
     $||\; r\_ := update\_refresh\_reg(r\_)$
  END
END;

RET =
BEGIN

$pc := bv16\_ushort(\ \ byte\_bv16(\ mem(sp\_plus\_one)\ ,\ bv\_9SP0\ ))\ ||\ sp := sp\_plus\_two\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

RET_cc(cc) =
PRE $cc \in 0\ ..\ 7$ **THEN**
IF $\ cc\_get(rgs8,cc) = 1$
THEN $pc := bv16\_ushort(\ \ byte\_bv16(\ mem(sp\_plus\_one)\ ,\ bv\_9SP0\ ))\ ||\ sp := sp\_plus\_two$
ELSE $pc := instruction\_next(pc)$ **END**
$||\ r\_ := update\_refresh\_reg(r\_)$
END;

RETI =
BEGIN
$pc := bv16\_ushort(\ \ byte\_bv16(\ mem(sp\_plus\_one)\ ,\ bv\_9SP0\ ))\ ||\ sp := sp\_plus\_two\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

RETN =
BEGIN
$pc := bv16\_ushort(\ \ byte\_bv16(\ mem(sp\_plus\_one)\ ,\ bv\_9SP0\ ))\ ||$
$sp := sp\_plus\_two\ ||\ r\_ := update\_refresh\_reg(r\_)\ ||\ iff1 := iff2$
END;

RST_p(pp) =
PRE $pp \in 0\ ..\ 7 \wedge bv16\_ushort(sp\_minus\_two) \in STACK\_R\_ADR\ \ \wedge$
$bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
ANY pc_l, pc_h **WHERE** $pc\_l \in BYTE \wedge pc\_h \in BYTE \wedge bv16\_byte(ushort\_bv16(pc)) = (pc\_l,pc\_h)$
THEN
updateStack($\{\ sp\_minus\_one \mapsto pc\_h,\ sp\_minus\_two \mapsto pc\_l\ \}$) $||$
$sp := sp\_minus\_two\ ||\ pc := pp \times 8\ \ \ \ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

AND_A_r(rr)=
PRE $rr \in id\_reg\_8$
THEN
ANY
$result\ \ ,\ negative\ ,\ zero\ ,\ half\_carry\ ,\ pv\ ,\ add\_sub\ ,\ carry$
WHERE $\ result \in BYTE \wedge negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT$
$\wedge\ pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$result = and(\ \ rgs8(a0),rgs8(rr)\ ) \wedge$
$negative = is\_negative(result) \wedge$
$zero = is\_zero(result)\ \ \ \wedge$
$half\_carry = 0 \wedge$
$pv = \ \ \ parity\_even(result)\ \ \ \ \ \wedge$
$add\_sub = \ \ 0 \wedge$
$carry = 0$
THEN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto result,$
$update\_flag\_reg(\ negative,zero,\ half\_carry,pv,add\_sub,\ carry)\ \ \ \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

```
AND_A_n(n1)=
PRE n1 ∈ SCHAR
THEN
    ANY
        result , negative , zero , half_carry , pv , add_sub , carry
    WHERE result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
        pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
        result = and( rgs8(a0), schar_byte( n1 )) ∧
        negative = is_negative(result) ∧
        zero = is_zero(result) ∧
        half_carry = 0 ∧
        pv = parity_even(result) ∧
        add_sub = 0 ∧
        carry = 0
    THEN
        rgs8:= rgs8 ⩤ { a0 ↦ result,
            update_flag_reg( negative,zero, half_carry,pv,add_sub, carry) } ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

AND_A_9HL0=
ANY
    result , negative , zero , half_carry , pv , add_sub , carry
WHERE result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
    pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
    result = and( rgs8(a0), bv_9HL0 ) ∧
    negative = is_negative(result) ∧
    zero = is_zero(result) ∧
    half_carry = 0 ∧
    pv = parity_even(result) ∧
    add_sub = 0 ∧
    carry = 0
THEN
    rgs8:= rgs8 ⩤ { a0 ↦ result,
        update_flag_reg( negative,zero, half_carry,pv,add_sub, carry) } ||
    pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
END ;

AND_A_9IX_d0(desloc)=
PRE desloc ∈ SCHAR THEN
    ANY
        result , negative , zero , half_carry , pv , add_sub , carry
    WHERE result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
        pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
        result = and( rgs8(a0), bv_9ireg_plus_d0(mem,ix,desloc) ) ∧
        negative = is_negative(result) ∧
        zero = is_zero(result) ∧
        half_carry = 0 ∧
        pv = parity_even(result) ∧
        add_sub = 0 ∧
        carry = 0
    THEN
        rgs8:= rgs8 ⩤ { a0 ↦ result,
```

```
        update_flag_reg( negative,zero, half_carry,pv,add_sub, carry)    } ||
      pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
  END
END;

AND_A_9IY_d0(desloc)=
PRE desloc ∈ SCHAR THEN
  ANY
    result  , negative , zero , half_carry , pv , add_sub , carry
  WHERE  result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
    pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
    result = and(  rgs8(a0), bv_9ireg_plus_d0(mem,iy,desloc) ) ∧
    negative = is_negative(result) ∧
    zero = is_zero(result)   ∧
    half_carry = 0 ∧
    pv =   parity_even(result)        ∧
    add_sub =  0 ∧
    carry = 0
  THEN
    rgs8:= rgs8 <+ { a0 ↦ result,
        update_flag_reg( negative,zero, half_carry,pv,add_sub, carry)    } ||
      pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
  END
END;

OR_A_r(rr)=
PRE rr ∈ id_reg_8
THEN
  ANY
    result  , negative , zero , half_carry , pv , add_sub , carry
  WHERE  result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT
     ∧ pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
    result = ior(rgs8(a0),rgs8(rr) ) ∧
    negative = is_negative(result) ∧
    zero = is_zero(result)   ∧
    half_carry = 0 ∧
    pv =   parity_even(result)        ∧
    add_sub =  0 ∧
    carry = 0
  THEN
    rgs8:= rgs8 <+ { a0 ↦ result,
        update_flag_reg( negative,zero, half_carry,pv,add_sub, carry)    } ||
      pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
  END
END;

OR_A_n(n1)=
PRE n1 ∈ SCHAR
THEN
  ANY
    result  , negative , zero , half_carry , pv , add_sub , carry
  WHERE  result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT
     ∧ pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
    result = ior( rgs8(a0), schar_byte( n1 )) ∧
```

$negative = is\_negative(result) \wedge$
$zero = is\_zero(result) \quad \wedge$
$half\_carry = 0 \wedge$
$pv = \quad parity\_even(result) \qquad \wedge$
$add\_sub = \quad 0 \wedge$
$carry = 0$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto result,$
$update\_flag\_reg(negative,zero, half\_carry,pv,add\_sub, carry) \quad \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

OR_A_9HL0=
ANY
$result \ , negative \ , zero \ , half\_carry \ , pv \ , add\_sub \ , carry$
WHERE $result \in BYTE \wedge negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge$
$pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$result = ior(\quad rgs8(a0), \quad bv\_9HL0) \wedge$
$negative = is\_negative(result) \wedge$
$zero = is\_zero(result) \quad \wedge$
$half\_carry = 0 \wedge$
$pv = \quad parity\_even(result) \qquad \wedge$
$add\_sub = \quad 0 \wedge$
$carry = 0$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto result,$
$update\_flag\_reg(negative,zero, half\_carry,pv,add\_sub, carry) \quad \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

OR_A_9IX_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$result \ , negative \ , zero \ , half\_carry \ , pv \ , add\_sub \ , carry$
WHERE $result \in BYTE \wedge negative \in BIT \wedge zero \in BIT \wedge half\_carry \in BIT \wedge$
$pv \in BIT \wedge add\_sub \in BIT \wedge carry \in BIT \wedge$
$result = ior(\quad rgs8(a0), \ bv\_9ireg\_plus\_d0(mem,ix,desloc)) \wedge$
$negative = is\_negative(result) \wedge$
$zero = is\_zero(result) \quad \wedge$
$half\_carry = 0 \wedge$
$pv = \quad parity\_even(result) \qquad \wedge$
$add\_sub = \quad 0 \wedge$
$carry = 0$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto result,$
$update\_flag\_reg(negative,zero, half\_carry,pv,add\_sub, carry) \quad \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

OR_A_9IY_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY

result , *negative* , *zero* , *half_carry* , *pv* , *add_sub* , *carry*
WHERE *result* ∈ *BYTE* ∧ *negative* ∈ *BIT* ∧ *zero* ∈ *BIT* ∧ *half_carry* ∈ *BIT* ∧
pv ∈ *BIT* ∧ *add_sub* ∈ *BIT* ∧ *carry* ∈ *BIT* ∧
result = *ior*(*rgs8*(*a0*), *bv_9ireg_plus_d0*(*mem*,*iy*,*desloc*)) ∧
negative = *is_negative*(*result*) ∧
zero = *is_zero*(*result*) ∧
half_carry = 0 ∧
pv = *parity_even*(*result*) ∧
add_sub = 0 ∧
carry = 0
THEN
rgs8:= *rgs8* ⩤ { *a0* ↦ *result*,
update_flag_reg(*negative*,*zero*, *half_carry*,*pv*,*add_sub*, *carry*) } ||
pc := *instruction_next*(*pc*) || *r_* := *update_refresh_reg*(*r_*)
END
END;

XOR_A_r(*rr*)=
PRE *rr* ∈ *id_reg_8*
THEN
ANY
result , *negative* , *zero* , *half_carry* , *pv* , *add_sub* , *carry*
WHERE *result* ∈ *BYTE* ∧ *negative* ∈ *BIT* ∧ *zero* ∈ *BIT* ∧ *half_carry* ∈ *BIT* ∧
pv ∈ *BIT* ∧ *add_sub* ∈ *BIT* ∧ *carry* ∈ *BIT* ∧
result = *xor*(*rgs8*(*a0*),*rgs8*(*rr*)) ∧
negative = *is_negative*(*result*) ∧
zero = *is_zero*(*result*) ∧
half_carry = 0 ∧
pv = *parity_even*(*result*) ∧
add_sub = 0 ∧
carry = 0
THEN
rgs8:= *rgs8* ⩤ { *a0* ↦ *result*,
update_flag_reg(*negative*,*zero*, *half_carry*,*pv*,*add_sub*, *carry*) } ||
pc := *instruction_next*(*pc*) || *r_* := *update_refresh_reg*(*r_*)
END
END;

XOR_A_n(*n1*)=
PRE *n1* ∈ *SCHAR*
THEN
ANY
result , *negative* , *zero* , *half_carry* , *pv* , *add_sub* , *carry*
WHERE *result* ∈ *BYTE* ∧ *negative* ∈ *BIT* ∧ *zero* ∈ *BIT* ∧ *half_carry* ∈ *BIT*
∧ *pv* ∈ *BIT* ∧ *add_sub* ∈ *BIT* ∧ *carry* ∈ *BIT* ∧
result = *xor*(*rgs8*(*a0*), *schar_byte*(*n1*)) ∧
negative = *is_negative*(*result*) ∧
zero = *is_zero*(*result*) ∧
half_carry = 0 ∧
pv = *parity_even*(*result*) ∧
add_sub = 0 ∧
carry = 0
THEN
rgs8:= *rgs8* ⩤ { *a0* ↦ *result*,

```
            update_flag_reg( negative,zero, half_carry,pv,add_sub, carry)    } ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

XOR_A_9HL0=
ANY
    result  , negative , zero , half_carry , pv , add_sub , carry
WHERE  result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
    pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
    result = xor(   rgs8(a0),   bv_9HL0 ) ∧
    negative = is_negative(result) ∧
    zero = is_zero(result)    ∧
    half_carry = 0 ∧
    pv =    parity_even(result)         ∧
    add_sub =   0 ∧
    carry = 0
THEN
    rgs8:= rgs8 <<+ { a0 ↦ result,
        update_flag_reg( negative,zero, half_carry,pv,add_sub, carry)    } ||
    pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
END;

XOR_A_9IX_d0(desloc)=
PRE desloc ∈ SCHAR THEN
    ANY
        result  , negative , zero , half_carry , pv , add_sub , carry
    WHERE  result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
        pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
        result = xor(   rgs8(a0), bv_9ireg_plus_d0(mem,ix,desloc) ) ∧
        negative = is_negative(result) ∧
        zero = is_zero(result)    ∧
        half_carry = 0 ∧
        pv =    parity_even(result)         ∧
        add_sub =   0 ∧
        carry = 0
    THEN
        rgs8:= rgs8 <<+ { a0 ↦ result,
            update_flag_reg( negative,zero, half_carry,pv,add_sub, carry)    } ||
        pc := instruction_next(pc) || r_ := update_refresh_reg(r_)
    END
END;

XOR_A_9IY_d0(desloc)=
PRE desloc ∈ SCHAR THEN
    ANY
        result  , negative , zero , half_carry , pv , add_sub , carry
    WHERE  result ∈ BYTE ∧ negative ∈ BIT ∧ zero ∈ BIT ∧ half_carry ∈ BIT ∧
        pv ∈ BIT ∧ add_sub ∈ BIT ∧ carry ∈ BIT ∧
        result = xor(   rgs8(a0), bv_9ireg_plus_d0(mem,iy,desloc) ) ∧
        negative = is_negative(result) ∧
        zero = is_zero(result)    ∧
        half_carry = 0 ∧
        pv =    parity_even(result)         ∧
```

$add\_sub = 0 \wedge$
$carry = 0$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto result,$
$update\_flag\_reg(negative, zero, half\_carry, pv, add\_sub, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

CP_A_r$(rr)=$
PRE $rr \in id\_reg\_8$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(rgs8(rr)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

CP_A_n$(n1)=$
PRE $n1 \in SCHAR$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0, byte\_uchar(rgs8(a0)), schar\_uchar(n1))$
THEN
$rgs8 := rgs8 <\!\!+ \{ update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

CP_A_9HL0$=$
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9HL0))$
THEN
$rgs8 := rgs8 <\!\!+ \{ update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

CP_A_9IX_d0$(desloc)=$
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero =$
$substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, ix, desloc)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$

$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

CP_A_9IY_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ =
$substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, iy, desloc)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

INC_r(rr) =
PRE
$rr \in id\_reg\_8 \wedge rr \neq f0$
THEN
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ = $add8UCHAR(0, byte\_uchar(rgs8(rr)), 1)$
THEN
$rgs8 := rgs8 <\!\!+ \{(rr \mapsto uchar\_byte(sum)),$
$update\_flag\_reg(negative, zero, half\_carry, carry, 0, carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

INC_9HL0 =
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ = $add8UCHAR(0, byte\_uchar(bv\_9HL0), 1)$
THEN
$rgs8 := rgs8 <\!\!+ \{update\_flag\_reg(negative, zero, half\_carry, carry, 0, carry) \}$ ||
updateAddressMem(bv_HL, $uchar\_byte(sum)$) ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

INC_9IX_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix, desloc)) \in DATA\_R\_ADR$ **THEN**
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ = $add8UCHAR(0, byte\_uchar(bv\_9ireg\_plus\_d0(mem, ix, desloc)), 1)$
THEN
$rgs8 := rgs8 <\!\!+ \{update\_flag\_reg(negative, zero, half\_carry, carry, 0, carry) \}$ ||
updateAddressMem($bv\_ireg\_plus\_d(ix, desloc)$, $uchar\_byte(sum)$) ||

$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

INC_9IY_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(0,byte\_uchar(bv\_9ireg\_plus\_d0(mem,iy,desloc)),1)$
THEN
$rgs8 := rgs8 \Leftarrow \{update\_flag\_reg(\ negative,zero,\ half\_carry,carry,0,\ carry)\ \}$ ||
updateAddressMem($bv\_ireg\_plus\_d(iy,desloc),\ uchar\_byte(sum)$) ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

DEC_r(rr) =
PRE
$rr \in id\_reg\_8 \wedge rr \neq f0$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0,byte\_uchar(\ rgs8(rr)),\ 1)$
THEN
$rgs8 := rgs8 \Leftarrow \{(rr \mapsto uchar\_byte(sum)\ ),$
$update\_flag\_reg(\ negative,zero,\ half\_carry,carry,1,\ z\_c)\ \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

DEC_9HL0 =
PRE $bv16\_ushort(bv\_HL) \in DATA\_R\_ADR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0,byte\_uchar(bv\_9HL0),1)$
THEN
$rgs8 := rgs8 \Leftarrow \{update\_flag\_reg(\ negative,zero,\ half\_carry,carry,1,\ z\_c)\ \}$ ||
updateAddressMem($\ bv\_HL\ ,\ uchar\_byte(sum)\ )$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

DEC_9IX_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(ix,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0,byte\_uchar(bv\_9ireg\_plus\_d0(mem,ix,desloc)),1)$
THEN
$rgs8 := rgs8 \Leftarrow \{update\_flag\_reg(\ negative,zero,\ half\_carry,carry,1,\ z\_c)\ \}$ ||
updateAddressMem($bv\_ireg\_plus\_d(ix,desloc),\ uchar\_byte(sum)$) ||

$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

DEC_9IY_d0($desloc$) =
PRE $desloc \in SCHAR \wedge bv16\_ushort(bv\_ireg\_plus\_d(iy,desloc)) \in DATA\_R\_ADR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0,byte\_uchar(bv\_9ireg\_plus\_d0(mem,iy,desloc)),1)$
THEN
$rgs8 := rgs8 <\!\!+ \{update\_flag\_reg(\ negative,zero,\ half\_carry,carry,1,\ z\_c)\ \}$ ||
updateAddressMem($bv\_ireg\_plus\_d(iy,desloc),\ uchar\_byte(sum)$) ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADD_HL_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg\ (ss = AF)$ **THEN**
ANY $result,\ bv\_value,\ bvh,\ bvl,$
$negative,\ carry,\ half\_carry,\ zero$
WHERE
$result \in USHORT \wedge bv\_value \in BV16 \quad \wedge \quad bvh \in BYTE \wedge \quad bvl \in BYTE \wedge$
$negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$get\_bv\_reg16(sp,rgs8,ss) = bv\_value \wedge$
$result = add16USHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ ) \wedge$
$bvh,bvl = bv16\_byte(ushort\_bv16(result)) \wedge$
$negative = z\_s \wedge$
$zero = z\_z \wedge$
$half\_carry = add\_halfcarryUSHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ \ ) \wedge$
$carry = \ \ add\_carryUSHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ \ )$
THEN
$rgs8 := rgs8 <\!\!+ \{\ h0 \mapsto bvh,\ \ l0 \mapsto bvl\ ,$
$update\_flag\_reg(\ negative,zero,\ half\_carry,z\_p\ ,0,\ carry)\ \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADC_HL_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg\ (ss = AF)$ **THEN**
ANY $result,\ bv\_value,\ bvh,\ bvl,$
$negative,\ carry,\ half\_carry,\ zero$
WHERE
$result \in USHORT \wedge bv\_value \in BV16 \quad \wedge \quad bvh \in BYTE \wedge \quad bvl \in BYTE \wedge$
$negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$get\_bv\_reg16(sp,rgs8,ss) = bv\_value \wedge$
$result = add16USHORT(z\_c\ ,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )) \wedge$
$negative = 1 \wedge$
$zero = is\_zero16USHORT(result) \wedge$
$half\_carry = add\_halfcarryUSHORT(z\_c\ ,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ \ ) \wedge$
$carry = \ \ add\_carryUSHORT(z\_c\ ,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ )$
THEN
$rgs8 := rgs8 <\!\!+ \{\ h0 \mapsto bvh,\ \ l0 \mapsto bvl\ ,$
$update\_flag\_reg(\ negative,zero,\ half\_carry,\ carry\ ,0,\ carry)\ \}$ ||

$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

SBC_HL_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg (ss = AF)$ **THEN**
ANY $result$, bv_value, bvh, bvl,
$negative$, $carry$, $half\_carry$, $zero$
WHERE
$result \in USHORT \wedge bv\_value \in BV16 \quad \wedge \quad bvh \in BYTE \wedge \quad bvl \in BYTE \wedge$
$negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$get\_bv\_reg16(sp,rgs8,ss) = bv\_value \wedge$
$result = sub16USHORT(z\_c,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )) \wedge$
$negative = 1 \wedge$
$zero = is\_zero16USHORT(result) \wedge$
$half\_carry = sub\_halfcarryUSHORT(z\_c,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ ) \wedge$
$carry = \quad sub\_carryUSHORT(z\_c,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\quad)$
THEN
$rgs8 := rgs8 \Leftarrow \{\ h0 \mapsto bvh,\quad l0 \mapsto bvl\ ,$
$update\_flag\_reg(\ negative,zero,\ half\_carry,\ carry\ ,0,\ carry)\ \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADD_IX_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg (ss = AF)$ **THEN**
ANY $result$, bv_value, bvh, bvl,
$negative$, $carry$, $half\_carry$, $zero$
WHERE
$result \in USHORT \wedge bv\_value \in BV16 \quad \wedge \quad bvh \in BYTE \wedge \quad bvl \in BYTE \wedge$
$negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$get\_bv\_reg16(sp,rgs8,ss) = bv\_value \wedge$
$result = add16USHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ ) \wedge$
$bvh,bvl = bv16\_byte(ushort\_bv16(result)) \wedge$
$negative = z\_s \wedge$
$zero = z\_z \wedge$
$half\_carry = add\_halfcarryUSHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\quad) \wedge$
$carry = \quad add\_carryUSHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\quad)$
THEN
$rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg(\ negative,zero,\ half\_carry,z\_p\ ,0,\ carry)\ \}$ ||
$ix := \quad ushort\_bv16(result)$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADD_IY_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg (ss = AF)$ **THEN**
ANY $result$, bv_value, bvh, bvl,
$negative$, $carry$, $half\_carry$, $zero$
WHERE
$result \in USHORT \wedge bv\_value \in BV16 \quad \wedge \quad bvh \in BYTE \wedge \quad bvl \in BYTE \wedge$
$negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$get\_bv\_reg16(sp,rgs8,ss) = bv\_value \wedge$
$result = add16USHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ ) \wedge$

$bvh,bvl = bv16\_byte(ushort\_bv16(result)) \wedge$
$negative = z\_s \wedge$
$zero = z\_z \wedge$
$half\_carry = add\_halfcarryUSHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ \ ) \wedge$
$carry = \ \ add\_carryUSHORT(0,\ bv16\_ushort(\ bv\_HL)\ ,\ bv16\_ushort(\ bv\_value\ )\ \ )$
THEN
$rgs8 := rgs8 \Leftarrow \{\ update\_flag\_reg(\ negative,zero,\ half\_carry,z\_p\ ,0,\ carry)\ \}\ ||$
$iy := \ \ ushort\_bv16(result)\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

INC_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg\ (ss = AF)$ **THEN**
IF $ss = SP$ **THEN** $sp := inc\_BV16(sp)$
ELSE
ANY
$rh,rl,\ vh,vl$
WHERE $rh \in id\_reg\_8 \wedge rl \in id\_reg\_8 \wedge vh \in BYTE \wedge \quad vl \in BYTE \wedge$
$REG16\_TO\_REG8(ss) = rh,\ rl \wedge \neg\ (\ rh = rl) \wedge$
$bv16\_byte(\ inc\_BV16(\ byte\_bv16(\ rgs8(rh),rgs8(rl)\ ))) = vh,vl$
THEN
$rgs8 := rgs8 \Leftarrow \{\ rh \mapsto vh\ ,\ rl \mapsto vl\ \}$
END
END
$||\qquad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

INC_IX=
BEGIN
$ix := inc\_BV16(ix)\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

INC_IY=
BEGIN
$iy := inc\_BV16(iy)\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

DEC_ss(ss)=
PRE $ss \in id\_reg\_16 \wedge \neg\ (ss = AF)$ **THEN**
IF $ss = SP$ **THEN** $sp := inc\_BV16(sp)$
ELSE
ANY
$rh,rl,\ vh,vl$
WHERE $rh \in id\_reg\_8 \wedge rl \in id\_reg\_8 \wedge vh \in BYTE \wedge \quad vl \in BYTE \wedge$
$REG16\_TO\_REG8(ss) = rh,\ rl \wedge \neg\ (\ rh = rl) \wedge$
$bv16\_byte(\ dec\_BV16(\ byte\_bv16(\ rgs8(rh),rgs8(rl)\ ))) = vh,vl$
THEN
$rgs8 := rgs8 \Leftarrow \{\ rh \mapsto vh\ ,\ rl \mapsto vl\ \}\qquad ||\qquad pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END
END;

DEC_IX=
BEGIN
$ix := dec\_BV16(ix)$ || $pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

DEC_IY=
BEGIN
$ix := dec\_BV16(ix)$ || $pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END

ADD_A_r(rr)=
PRE $rr \in id\_reg\_8$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(0, byte\_uchar(\ rgs8(a0)), byte\_uchar(\ rgs8(rr))\ ) \wedge$
dom($add8UCHAR$) = $BIT \times UCHAR \times UCHAR$
THEN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto uchar\_byte(sum)\ ,$
$(update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 0,\ carry))$ }||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADD_A_n($n1$)=
PRE $n1 \in SCHAR$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(0, byte\_uchar(rgs8(a0)),\ schar\_uchar(\ n1))$
THEN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 0,\ carry)\ \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADD_A_9HL0=
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9HL0))$
THEN
$rgs8 := rgs8 \Leftarrow \{\ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 0,\ carry)\ \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

ADD_A_9IX_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$

$sum, negative, carry, half\_carry, zero =$
$add8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem,ix,desloc)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative,zero, half\_carry,carry,0, carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADD_A_9IY_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero =$
$add8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem,iy,desloc)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative,zero, half\_carry,carry,0, carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADC_A_r(rr)=
PRE $rr \in id\_reg\_8$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar(rgs8(rr)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative,zero, half\_carry,carry,0, carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADC_A_n($n1$)=
PRE $n1 \in SCHAR$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(z\_c, byte\_uchar(rgs8(a0)), schar\_uchar(n1))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative,zero, half\_carry,carry,0, carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADC_A_9HL0=
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = add8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9HL0))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative,zero, half\_carry,carry,0, carry) \}$ ||

$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END;

ADC_A_9IX_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ =
$add8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, ix, desloc)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(negative, zero,\ half\_carry, carry, 0,\ carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

ADC_A_9IY_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ =
$add8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, iy, desloc)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(negative, zero,\ half\_carry, carry, 0,\ carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

SUB_A_r(rr)=
PRE $rr \in id\_reg\_8$
THEN
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ = $substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(rgs8(rr)))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(negative, zero,\ half\_carry, carry, 1,\ carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

SUB_A_n($n1$)=
PRE $n1 \in SCHAR$
THEN
ANY
sum, $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum, $negative$, $carry$, $half\_carry$, $zero$ = $substract8UCHAR(0, byte\_uchar(rgs8(a0)),\ schar\_uchar(\ n1))$
THEN
$rgs8 := rgs8 <\!\!+ \{ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(negative, zero,\ half\_carry, carry, 1,\ carry) \}$ ||
$pc := instruction\_next(pc)$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

SUB_A_9HL0=
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9HL0))$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

SUB_A_9IX_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero =$
$substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, ix, desloc)))$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

SUB_A_9IY_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero =$
$substract8UCHAR(0, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, iy, desloc)))$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

SBC_A_r(rr)=
PRE $rr \in id\_reg\_8$
THEN
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar(rgs8(rr)))$
THEN
$rgs8 := rgs8 \Leftarrow \{ a0 \mapsto uchar\_byte(sum), update\_flag\_reg(negative, zero, half\_carry, carry, 1, carry) \} \parallel$
$pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END
END;

SBC_A_n($n1$)=
PRE $n1 \in SCHAR$
THEN
ANY

$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(z\_c, byte\_uchar(rgs8(a0)), schar\_uchar(\ n1\ )\ )$
THEN
$rgs8 := rgs8 \leftarrow\!\!\!+ \{\ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 1,\ carry)\ \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

SBC_A_9HL0=
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero = substract8UCHAR(z\_c,\ byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9HL0\ ))$
THEN
$rgs8 := rgs8 \leftarrow\!\!\!+ \{\ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 1,\ carry)\ \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

SBC_A_9IX_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero =$
$substract8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar((bv\_9ireg\_plus\_d0(mem, ix, desloc)\ )))$
THEN
$rgs8 := rgs8 \leftarrow\!\!\!+ \{\ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 1,\ carry)\ \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

SBC_A_9IY_d0($desloc$)=
PRE $desloc \in SCHAR$ **THEN**
ANY
$sum, negative, carry, half\_carry, zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
$sum, negative, carry, half\_carry, zero =$
$substract8UCHAR(z\_c, byte\_uchar(rgs8(a0)), byte\_uchar(bv\_9ireg\_plus\_d0(mem, ix, desloc)\ ))$
THEN
$rgs8 := rgs8 \leftarrow\!\!\!+ \{\ a0 \mapsto uchar\_byte(sum),\ update\_flag\_reg(\ negative, zero,\ half\_carry, carry, 1,\ carry)\ \}\ ||$
$pc := instruction\_next(pc)\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END

CPI =
ANY
$sum\ ,\ negative\ ,\ carry\ ,\ half\_carry\ ,\ zero\ ,$
$hvn\ ,\ lvn\ ,\ bvn\ ,\ cvn$
WHERE
$sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \quad \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge bvn \in BYTE \wedge cvn \in BYTE \wedge$
$sum\ ,\ negative\ ,\ carry\ ,\ half\_carry\ ,\ zero =$
$substract8UCHAR\ (\ 0\ ,\ byte\_uchar\ (\ rgs8\ (\ a0\ )\ )\ ,\ byte\_uchar\ (\ bv\_9HL0\ )\ ) \wedge$

hvn , $lvn = bv16\_byte$ (inc_BV16 (bv_HL)) $\wedge$
bvn , $cvn = bv16\_byte$ (dec_BV16 (bv_BC))
THEN
IF $zero = 1$ **THEN**
$rgs8 := rgs8 \Leftarrow \{ h0 \mapsto hvn$, $l0 \mapsto lvn$, $b0 \mapsto bvn$, $c0 \mapsto cvn$
, $update\_flag\_reg($ $negative$, $zero$, $half\_carry$,
bit_not ($is\_zero16USHORT$ ($bv16\_ushort$ ($dec\_BV16(bv\_BC)$))) , 1 , z_c) $\}$
END ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

CPIR =
ANY
sum , $negative$, $carry$, $half\_carry$, $zero$,
hvn , lvn , bvn , cvn
WHERE
$sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \quad \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge bvn \in BYTE \wedge cvn \in BYTE \wedge$
$substract8UCHAR$ (0 , $byte\_uchar$ ($rgs8$ ($a0$)) , $byte\_uchar$ (bv_9HL0)) =
sum , $negative$, $carry$, $half\_carry$, $zero \wedge$
hvn , $lvn = bv16\_byte$ (inc_BV16 (bv_HL)) $\wedge$
bvn , $cvn = bv16\_byte$ (dec_BV16 (bv_BC))
THEN
IF $zero = 1$ **THEN**
$rgs8 := rgs8 \Leftarrow \{ h0 \mapsto hvn$, $l0 \mapsto lvn$, $b0 \mapsto bvn$, $c0 \mapsto cvn$
, $update\_flag\_reg($ $negative$, $zero$, $half\_carry$
, bit_not ($is\_zero16USHORT$ ($bv16\_ushort$ ($dec\_BV16(bv\_BC)$))) , 1 , z_c) $\}$
END
|| $r\_ := update\_refresh\_reg(r\_)$ ||
IF $is\_zero16USHORT$ ($bv16\_ushort$ ($dec\_BV16(bv\_BC)$)) = $0 \vee (zero = 1)$
THEN $pc := instruction\_next$ (pc) **END**
END;

CPD =
ANY
sum , $negative$, $carry$, $half\_carry$, $zero$,
hvn , lvn , bvn , cvn
WHERE
$sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \quad \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge bvn \in BYTE \wedge cvn \in BYTE \wedge$
$substract8UCHAR$ (0 , $byte\_uchar$ ($rgs8$ ($a0$)) , $byte\_uchar$ (bv_9HL0)) =
sum , $negative$, $carry$, $half\_carry$, $zero \wedge$
hvn , $lvn = bv16\_byte$ (dec_BV16 (bv_HL)) $\wedge$
bvn , $cvn = bv16\_byte$ (dec_BV16 (bv_BC))
THEN
IF $zero = 1$ **THEN**
$rgs8 := rgs8 \Leftarrow \{ h0 \mapsto hvn$, $l0 \mapsto lvn$, $b0 \mapsto bvn$, $c0 \mapsto cvn$
, $update\_flag\_reg($ $negative$, $zero$, $half\_carry$
, bit_not ($is\_zero16USHORT$ ($bv16\_ushort$ ($dec\_BV16(bv\_BC)$))) , 1 , z_c) $\}$
END
||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

CPDR =
ANY
sum , $negative$, $carry$, $half\_carry$, $zero$,
hvn , lvn , bvn , cvn
WHERE
$sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \quad \wedge$
$hvn \in BYTE \wedge lvn \in BYTE \wedge bvn \in BYTE \wedge cvn \in BYTE \wedge$
$substract8UCHAR\ (\ 0\ ,\ byte\_uchar\ (\ rgs8\ (\ a0\ )\ )\ ,\ byte\_uchar\ (\ bv\_9HL0\ )\ ) =$
sum , $negative$, $carry$, $half\_carry$, $zero \wedge$
hvn , $lvn = bv16\_byte\ (\ dec\_BV16\ (\ bv\_HL\ )\ ) \wedge$
bvn , $cvn = bv16\_byte\ (\ dec\_BV16\ (\ bv\_BC\ )\ )$
THEN
IF $zero = 1$ **THEN**
$rgs8 := rgs8 \leftarrow\!\!\!+ \{\ h0 \mapsto hvn\ ,\ l0 \mapsto lvn\ ,\ b0 \mapsto bvn\ ,\ c0 \mapsto cvn$
$,\ \ update\_flag\_reg(\ negative\ ,\ zero\ ,\ half\_carry$
$,\ bit\_not\ (\ is\_zero16USHORT\ (\ bv16\_ushort\ (\ dec\_BV16(bv\_BC)\ \ )\ )\ ,\ 1\ ,\ z\_c\ )\ \}$
END ||
IF $is\_zero16USHORT\ (\ bv16\_ushort\ (\ dec\_BV16(bv\_BC)\ \ )\ ) = 0 \vee (\ zero = 1\ )$
THEN $pc := instruction\_next\ (\ pc\ )$ **END** ||
$r\_ := update\_refresh\_reg(r\_)$
END;

DAA =
ANY
$result$, $s0$, $z0$,$h0$,$pv0$, $n0$, $c0$
WHERE
$result \in BYTE \wedge s0 \in BIT \wedge z0 \in BIT \wedge h0 \in BIT \wedge pv0 \in BIT \wedge n0 \in BIT \wedge c0 \in BIT \wedge$
$daa\_function(z\_n, z\_c\ , z\_h\ , rgs8\ (\ a0\ )\ ) = (\ result\ ,\ \ c0, h0) \wedge$
$s0 = bitget(result,7) \wedge$
$z0 = is\_zero(result) \wedge$
$pv0 = parity\_even(result) \wedge$
$n0 = z\_n$
THEN
$rgs8 :=\ \ rgs8 \leftarrow\!\!\!+ \{\ a0 \mapsto result\ ,\ \ update\_flag\_reg(\ s0, z0,h0,pv0, n0, c0\ \ )\ \}\ \ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

CPL =
ANY $result$
WHERE $result \in BYTE\ \ \wedge result = complement\ (\ rgs8\ (\ a0\ )\ )$
THEN
$rgs8 :=\ \ \ rgs8 \leftarrow\!\!\!+ \{\ \ (\ a0 \mapsto result\ )\ ,$
$update\_flag\_reg(\ z\_s\ ,\ z\_z\ ,\ 1\ ,\ z\_p\ ,\ 1\ ,\ z\_c\ )\ \}\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END ;

NEG =
ANY
sum , $negative$, $carry$, $half\_carry$, $zero$
WHERE $sum \in UCHAR \wedge negative \in BIT \wedge carry \in BIT \wedge half\_carry \in BIT \wedge zero \in BIT \wedge$
sum , $negative$, $carry$, $half\_carry$, $zero = substract8UCHAR\ (\ 0\ ,\ 0\ ,\ byte\_uchar\ (\ rgs8\ (\ a0\ )\ )\ )$
THEN
$rgs8 := rgs8 \leftarrow\!\!\!+ \{\ a0 \mapsto uchar\_byte\ (\ sum\ )\ ,\ \ update\_flag\_reg(\ negative\ ,\ zero\ ,\ half\_carry\ ,\ carry\ ,\ 1\ ,\ carry\ )\ \}\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$

END ;

CCF =
BEGIN
  $rgs8 := rgs8 \Leftarrow \{ \quad update\_flag\_reg(z\_s , z\_z , z\_h , z\_p , 0 , bit\_not (z\_c)) \} \parallel$
  $pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END ;

SCF =
BEGIN
  $rgs8 := rgs8 \Leftarrow \{ \quad update\_flag\_reg(z\_s , z\_z , z\_h , z\_p , 0 , 1) \} \parallel$
  $pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END ;

NOP =
BEGIN
  $pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END ;

HALT =
BEGIN
  $r\_ := update\_refresh\_reg(r\_)$
END ;

DI=
BEGIN
  $iff1 := 0 \parallel iff2 := 0 \quad \parallel \quad pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

EI=
BEGIN
  $iff1 := 1 \parallel iff2 := 1 \quad \parallel \quad pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

IM0=
BEGIN
  $im := (0 \mapsto 0) \quad \parallel \quad pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

IM1=
BEGIN
  $im := (0 \mapsto 1) \quad \parallel \quad pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

IM2=
BEGIN
  $im := (1 \mapsto 1) \quad \parallel \quad pc := instruction\_next (pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

set_pc($value$)=
PRE $value \in USHORT$ **THEN**
  pc:=0
END;

result ← **get_pc** =
BEGIN
  result := *pc*
END;

LD_r_r_ (*rr* , *rr_*) =
PRE *rr* ∈ *id_reg_8* ∧ *rr_* ∈ *id_reg_8* **THEN**
  rgs8 (*rr*) := *rgs8* (*rr_*) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_r_n_ (*rr* , *n0*) =
PRE *rr* ∈ *id_reg_8* ∧ *n0* ∈ *SCHAR* **THEN**
  rgs8 (*rr*) := *schar_byte* (*n0*) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_r_9HL0 (*rr*) =
PRE *rr* ∈ *id_reg_8* **THEN**
  ANY *address* **WHERE** *address* ∈ *BV16* ∧
    address = *byte_bv16* (*rgs8* (*h0*) , *rgs8* (*l0*))
  THEN
    rgs8 (*rr*) := *mem* (*address*) ||
    pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
  END
END;

LD_r_9IX_d0 (*rr* , *desloc*) =
PRE *rr* ∈ *id_reg_8* ∧ *desloc* ∈ *SCHAR*
THEN
  rgs8 (*rr*) := *bv_9ireg_plus_d0* (*mem* , *ix* , *desloc*) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_r_9IY_d0 (*rr* , *desloc*) =
PRE *rr* ∈ *id_reg_8* ∧ *desloc* ∈ *SCHAR*
THEN
  rgs8 (*rr*) := *bv_9ireg_plus_d0* (*mem* , *iy* , *desloc*) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9HL0_r (*rr*) =
PRE *rr* ∈ *id_reg_8* ∧ *bv16_ushort*(*bv_HL*) ∈ *DATA_R_ADR* **THEN**
  updateAddressMem (*bv_HL* , *rgs8* (*rr*)) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9IX_d0_r (*desloc* , *rr*) =
PRE *desloc* ∈ *SCHAR* ∧ *rr* ∈ *id_reg_8* ∧ *bv16_ushort*(*bv_ireg_plus_d* (*ix* , *desloc*)) ∈ *DATA_R_ADR* **THEN**
  updateAddressMem (*bv_ireg_plus_d* (*ix* , *desloc*) , *rgs8* (*rr*)) ||
  pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9IY_d0_r (*desloc* , *rr*) =
PRE *desloc* ∈ *SCHAR* ∧ *rr* ∈ *id_reg_8* ∧ *bv16_ushort*(*bv_ireg_plus_d* (*iy* , *desloc*)) ∈ *DATA_R_ADR* **THEN**
updateAddressMem (*bv_ireg_plus_d* (*iy* , *desloc*) , *rgs8* (*rr*)) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9HL0_n (*n0*) =
PRE *n0* ∈ *SCHAR* ∧ *bv16_ushort*(*bv_HL*) ∈ *DATA_R_ADR* **THEN**
updateAddressMem (*bv_HL* , *schar_byte* (*n0*)) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9IX_d0_n (*desloc* , *n0*) =
PRE *desloc* ∈ *SCHAR* ∧ *n0* ∈ *SCHAR* ∧ *bv16_ushort*(*bv_ireg_plus_d* (*ix* , *desloc*)) ∈ *DATA_R_ADR* **THEN**
updateAddressMem (*bv_ireg_plus_d* (*ix* , *desloc*) , *schar_byte* (*n0*)) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9IY_d0_n (*desloc* , *n0*) =
PRE *desloc* ∈ *SCHAR* ∧ *n0* ∈ *SCHAR* ∧ *bv16_ushort*(*bv_ireg_plus_d* (*iy* , *desloc*)) ∈ *DATA_R_ADR* **THEN**
updateAddressMem (*bv_ireg_plus_d* (*iy* , *desloc*) , *schar_byte* (*n0*)) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_A_9BC0 =
BEGIN
rgs8 (*a0*) := *bv_9BC0* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_A_9DE0 =
BEGIN
rgs8 (*a0*) := *bv_9DE0* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_A_9nn0 (*nn*) =
PRE *nn* ∈ *USHORT*
THEN
rgs8 (*a0*) := *mem* (*ushort_bv16* (*nn*)) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9BC0_A =
PRE *bv16_ushort*(*bv_BC*) ∈ *DATA_R_ADR*
THEN
updateAddressMem (*bv_BC* , *rgs8* (*a0*)) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_9DE0_A =
PRE *bv16_ushort*(*bv_DE*) ∈ *DATA_R_ADR*
THEN

updateAddressMem ((bv_DE) , $rgs8$ ($a0$)) ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

LD_9nn0_A (nn) =
PRE $nn \in USHORT \quad \wedge \quad nn \quad \in DATA\_R\_ADR$
THEN
updateAddressMem ($ushort\_bv16$ (nn) , $rgs8$ ($a0$)) ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

LD_A_I=
BEGIN
$rgs8 := rgs8 \quad \Leftarrow \{ a0 \mapsto i\_ , \quad update\_flag\_reg($ $is\_negative(i\_)$, $is\_zero($ $i\_$ $)$, 0 , $iff2$, 0, z_c $)\}$ ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

LD_A_R=
BEGIN
$rgs8 := rgs8 \quad \Leftarrow \{ a0 \mapsto r\_ , \quad update\_flag\_reg($ $is\_negative($ $r\_$ $)$, $is\_zero($ $r\_$ $)$, 0 , $iff2$, 0, z_c $) \quad \}$ ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

LD_I_A=
BEGIN
$i\_ := rgs8(a0) \quad$ ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END;

LD_R_A=
BEGIN
$r\_ := rgs8(a0) \quad$ ||
$pc := instruction\_next$ (pc)
END;

LD_dd_nn (dd , nn) =
PRE $dd \in id\_reg\_16 \wedge nn \in USHORT \quad \wedge dd \neq AF$
THEN
IF $dd = SP$ **THEN** $sp := ushort\_bv16$ (nn)
ELSE
ANY rh , rl , $w1$, $w2$ **WHERE**
$rh \in id\_reg\_8 \wedge \quad rl \in id\_reg\_8 \wedge$
$w1 \in BYTE \wedge w2 \in BYTE \wedge$
rh , $rl = REG16\_TO\_REG8$ (dd) $\wedge \neg (rh = rl) \wedge$
$bv16\_byte$ ($ushort\_bv16$ (nn)) $= w1$, $w2$
THEN
$rgs8 := rgs8 \Leftarrow \{ rh \mapsto w1$, $rl \mapsto w2 \}$ ||
$pc := instruction\_next$ (pc) || $r\_ := update\_refresh\_reg(r\_)$
END
END
END;

LD_IX_nn (nn) =

PRE $nn \in USHORT$
THEN
$ix := ushort\_bv16\ (\ nn\ )\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

LD_IY_nn (nn) =
PRE $nn \in USHORT$
THEN
$iy := ushort\_bv16\ (\ nn\ )\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

LD_HL_9nn0 (nn) =
PRE $nn \in USHORT$
THEN
$rgs8 := rgs8 \Leftarrow \{\ h0 \mapsto mem\ (\ ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ )\ )\ ,$
$l0 \mapsto mem\ (\ ushort\_bv16\ (\ nn\ )\ )\ \}\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

LD_dd_9nn0 (dd , nn) =
PRE $dd \in id\_reg\_16 \wedge nn \in USHORT \quad \wedge dd \neq AF$
THEN
$r\_ := update\_refresh\_reg(r\_)\ ||$
IF $dd = SP$ **THEN**
$sp := byte\_bv16\ (\ mem\ (\ ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ )\ )\ ,\ mem\ (\ ushort\_bv16\ (\ nn\ )\ )\ )$
ELSE
ANY rh , rl , $w1$, $w2$ **WHERE**
$rh \in id\_reg\_8 \wedge \quad rl \in id\_reg\_8 \wedge$
$w1 \in BYTE \wedge w2 \in BYTE \wedge$
$rh\ ,\ rl = REG16\_TO\_REG8\ (\ dd\ ) \wedge \quad \neg\ (rh = rl) \wedge$
$w1 = mem\ (\ ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ )\ ) \wedge$
$w2 = mem\ (\ ushort\_bv16\ (\ nn\ )\ )$
THEN
$rgs8 := rgs8 \Leftarrow \{\ rh \mapsto w1\ ,\ rl \mapsto w2\ \}\ ||$
$pc := instruction\_next\ (\ pc\ )$
END
END
END;

LD_IX_9nn0 (nn) =
PRE $nn \in USHORT$
THEN
$ix := byte\_bv16\ (\ mem\ (\ ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ )\ )\ ,\ mem\ (\ ushort\_bv16\ (\ nn\ )\ )\ )\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

LD_IY_9nn0 (nn) =
PRE $nn \in USHORT$
THEN
$iy := byte\_bv16\ (\ mem\ (\ ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ )\ )\ ,\ mem\ (\ ushort\_bv16\ (\ nn\ )\ )\ )\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

LD_9nn0_HL (nn) =
PRE $nn \in USHORT \wedge$
$nn \in DATA\_R\_ADR \wedge add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ ) \in DATA\_R\_ADR$
THEN
updateMem ({ $ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ ) \mapsto rgs8\ (\ h0\ )$,
$ushort\_bv16\ (\ nn\ ) \mapsto rgs8\ (\ l0\ )$ }) ||
$pc := instruction\_next\ (\ pc\ )$ || $r\_ := update\_refresh\_reg(r\_)$
END;

LD_9nn0_dd (nn , dd) =
PRE $dd \in id\_reg\_16 \wedge dd \neq AF \wedge nn \in USHORT \wedge$
$nn \in DATA\_R\_ADR \wedge add16USHORT(\ 0\ ,\ nn\ ,\ 1\ ) \in DATA\_R\_ADR$
THEN
IF $dd = SP$
THEN
ANY vh , vl **WHERE**
$vh \in BYTE \wedge vl \in BYTE \wedge$
$bv16\_byte\ (\ sp\ ) = vh\ ,\ vl \wedge$
$\neg\ (ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ ) = ushort\_bv16\ (\ nn\ ))$
THEN
updateMem ({ $ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ ) \mapsto vh$,
$ushort\_bv16\ (\ nn\ ) \mapsto vl$ })
|| $r\_ := update\_refresh\_reg(r\_)$
END
ELSE
ANY rh , rl , $w1$, $w2$ **WHERE**
$rh \in id\_reg\_8 \wedge rl \in id\_reg\_8 \wedge$
$w1 \in SCHAR \wedge w2 \in BYTE \wedge$
$rh\ ,\ rl = REG16\_TO\_REG8\ (\ dd\ ) \wedge \neg\ (rh = rl)$
THEN
updateMem ({ $ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ ) \mapsto rgs8\ (\ rh\ )$,
$ushort\_bv16\ (\ nn\ ) \mapsto rgs8\ (\ rl\ )$ }) ||
$pc := instruction\_next\ (\ pc\ )$ || $r\_ := update\_refresh\_reg(r\_)$
END
END
END;

LD_9nn0_IX (nn) =
PRE $nn \in USHORT \wedge nn \in DATA\_R\_ADR \wedge add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ ) \in DATA\_R\_ADR$ **THEN**
ANY h_ix , l_ix **WHERE**
$h\_ix \in BYTE \wedge l\_ix \in BYTE \wedge$
$h\_ix\ ,\ l\_ix = bv16\_byte\ (\ ix\ )$
THEN
updateMem ({ $ushort\_bv16\ (\ add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ )\ ) \mapsto h\_ix$, $ushort\_bv16\ (\ nn\ ) \mapsto l\_ix$ }) ||
$pc := instruction\_next\ (\ pc\ )$ || $r\_ := update\_refresh\_reg(r\_)$
END
END;

LD_9nn0_IY (nn) =
PRE $nn \in USHORT \wedge nn \in DATA\_R\_ADR \wedge add16USHORT\ (\ 0\ ,\ nn\ ,\ 1\ ) \in DATA\_R\_ADR$ **THEN**
ANY h_iy , l_iy **WHERE**
$h\_iy \in BYTE \wedge l\_iy \in BYTE \wedge$
$h\_iy\ ,\ l\_iy = bv16\_byte\ (\ iy\ )$

THEN
updateMem ({ *ushort_bv16* (*add16USHORT* (0 , *nn* , 1)) ↦ *h_iy* , *ushort_bv16* (*nn*) ↦ *l_iy* }) ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END
END;

LD_SP_HL =
BEGIN
sp := *bv_HL* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_SP_IX =
BEGIN
sp := *ix* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

LD_SP_IY =
BEGIN
sp := *iy* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

PUSH_qq (*qq*) =
PRE *qq* ∈ *id_reg_16* ∧ *qq* ≠ *SP*
THEN
ANY
qqh , *qql*
WHERE *qqh* ∈ *id_reg_8* ∧ *qql* ∈ *id_reg_8* ∧
REG16_TO_REG8 (*qq*) = *qqh* , *qql* ∧ ¬ (*qqh* = *qql*) ∧
{ *sp_minus_two* ↦ *rgs8* (*qql*) ,
sp_minus_one ↦ *rgs8* (*qqh*) } ∈ *BV16* ⟶ *BYTE*
THEN
updateStack ({ *sp_minus_two* ↦ *rgs8* (*qql*) ,
sp_minus_one ↦ *rgs8* (*qqh*) }) ||
sp := *sp_minus_two* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END
END;

PUSH_IX =
ANY
wh , *wl*
WHERE *wh* ∈ *BYTE* ∧ *wl* ∈ *BYTE* ∧
bv16_byte (*ix*) = *wh* , *wl* ∧
{ *sp_minus_two* ↦ *wl* , *sp_minus_one* ↦ *wh* } ∈ *BV16* ⟶ *BYTE*
THEN
updateStack ({ *sp_minus_two* ↦ *wl* ,
sp_minus_one ↦ *wh* }) ||
sp := *sp_minus_two* ||
pc := *instruction_next* (*pc*) || *r_* := *update_refresh_reg*(*r_*)
END;

PUSH_IY =
ANY
wh , wl
WHERE $wh \in BYTE \wedge wl \in BYTE \wedge$
$bv16\_byte\ (\ iy\ ) = wh\ ,\ wl \wedge$
$\{\ sp\_minus\_two \mapsto wl\ ,\ sp\_minus\_one \mapsto wh\ \} \in BV16 \rightarrow BYTE$
THEN
updateStack ($\{\ sp\_minus\_two \mapsto wl\ ,$
$sp\_minus\_one \mapsto wh\ \}\ )\ ||$
$sp := sp\_minus\_two\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END;

POP_qq (qq) =
PRE $qq \in id\_reg\_16 \wedge qq \neq SP$
THEN
ANY
qqh , qql
WHERE $qqh \in id\_reg\_8 \wedge qql \in id\_reg\_8 \wedge$
$REG16\_TO\_REG8\ (\ qq\ ) = qqh\ ,\ qql \wedge \quad \neg\ (\ qqh = qql) \wedge$
$\{\ qql \mapsto mem\ (\ sp\_plus\_two\ )\ ,\ qqh \mapsto mem\ (\ sp\_plus\_one\ )\ \} \in id\_reg\_8 \rightarrow BYTE$
THEN
$rgs8 := rgs8 \Leftarrow \{\ qql \mapsto mem\ (\ sp\_plus\_two\ )\ ,\ qqh \mapsto mem\ (\ sp\_plus\_one\ )\ \}\ ||$
$sp := sp\_plus\_two\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

POP_IX =
PRE $sp \in BV16$
THEN
ANY
$bv16$
WHERE $bv16 \in BV16 \quad \wedge$
$byte\_bv16\ (\ mem\ (\ sp\_plus\_one\ )\ ,\ mem\ (\ sp\_plus\_two\ )\ ) = bv16$
THEN
$ix := bv16\ ||$
$sp := sp\_plus\_two\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

POP_IY =
PRE $sp \in BV16$
THEN
ANY
$bv16$
WHERE $bv16 \in BV16 \quad \wedge$
$byte\_bv16\ (\ mem\ (\ sp\_plus\_one\ )\ ,\ mem\ (\ sp\_plus\_two\ )\ ) = bv16$
THEN
$iy := bv16\ ||$
$sp := sp\_plus\_two\ ||$
$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END

END;

EX_DE_HL =
BEGIN
  $rgs8 := rgs8 \lhd\!\!\!- \{ d0 \mapsto rgs8(h0), e0 \mapsto rgs8(l0), h0 \mapsto rgs8(d0), l0 \mapsto rgs8(e0) \} \parallel$
  $pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

EX_AF_AF_ =
BEGIN
  $rgs8 := rgs8 \lhd\!\!\!- \{ a0 \mapsto rgs8(a\_0), f0 \mapsto rgs8(f\_0), a\_0 \mapsto rgs8(a0), f\_0 \mapsto rgs8(f0) \} \parallel$
  $pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

EXX =
BEGIN
  $rgs8 := rgs8 \lhd\!\!\!- \{ b0 \mapsto rgs8(b\_0), c0 \mapsto rgs8(c\_0), d0 \mapsto rgs8(d\_0), e0 \mapsto rgs8(e\_0),$
  $h0 \mapsto rgs8(h\_0), l0 \mapsto rgs8(l\_0), b\_0 \mapsto rgs8(b0), c\_0 \mapsto rgs8(c0),$
  $d\_0 \mapsto rgs8(d0), e\_0 \mapsto rgs8(e0), h\_0 \mapsto rgs8(h0), l\_0 \mapsto rgs8(l0) \} \parallel$
  $pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

EX_9SP0_HL =
PRE
  $bv16\_ushort(sp\_plus\_one) \in STACK\_R\_ADR \wedge bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
  $rgs8 := rgs8 \lhd\!\!\!- \{ h0 \mapsto mem(sp\_plus\_one), l0 \mapsto mem(sp) \} \parallel$
  updateStack $(\{ sp\_plus\_one \mapsto rgs8(h0), sp \mapsto rgs8(l0) \}) \parallel$
  $pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
END;

EX_9SP0_IX =
PRE $bv16\_ushort(sp\_plus\_one) \in STACK\_R\_ADR \wedge bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
  ANY wh, wl
  WHERE $wh \in BYTE \wedge wl \in BYTE \wedge$
    $bv16\_byte(ix) = wh, wl$
  THEN
    $ix := byte\_bv16(mem(sp\_plus\_one), mem(sp)) \parallel$
    updateStack $(\{ sp\_plus\_one \mapsto wh, sp \mapsto wl \}) \parallel$
    $pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$
  END
END;

EX_9SP0_IY =
PRE $bv16\_ushort(sp\_plus\_one) \in STACK\_R\_ADR \wedge bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
  ANY wh, wl
  WHERE $wh \in BYTE \wedge wl \in BYTE \wedge$
    $bv16\_byte(iy) = wh, wl$
  THEN
    $iy := byte\_bv16(mem(sp\_plus\_one), mem(sp)) \parallel$
    updateStack $(\{ sp\_plus\_one \mapsto wh, sp \mapsto wl \}) \parallel$
    $pc := instruction\_next(pc) \parallel r\_ := update\_refresh\_reg(r\_)$

```
    END
END;

LDI =
PRE bv16_ushort(bv_DE) ∈ DATA_R_ADR     THEN
    ANY   hvn , lvn , dvn , evn , bvn , cvn
    WHERE
        hvn , lvn = bv16_byte ( inc_BV16 ( bv_HL ) ) ∧
        dvn , evn = bv16_byte ( inc_BV16 ( bv_DE ) ) ∧
        bvn , cvn = bv16_byte ( dec_BV16 ( bv_BC ) )
    THEN
        updateAddressMem ( bv_DE , bv_9HL0 ) ||
        rgs8 :=   rgs8 <+ { ( h0 ↦ hvn ) , ( l0 ↦ lvn ) ,
            ( d0 ↦ dvn ) , ( e0 ↦ evn ) ,
            ( b0 ↦ bvn ) , ( c0 ↦ cvn ) ,
            update_flag_reg( z_s , z_z , 0 ,
            bit_not ( is_zero16USHORT ( bv16_ushort ( dec_BV16(bv_BC)   )   ) ) , 0 , z_c ) } ||
        pc := instruction_next ( pc ) || r_ := update_refresh_reg(r_)
    END
END;

LDIR =
PRE bv16_ushort(bv_DE) ∈ DATA_R_ADR     THEN
    ANY   hvn , lvn , dvn , evn , bvn , cvn
    WHERE
        hvn , lvn = bv16_byte ( inc_BV16 ( bv_HL ) ) ∧
        dvn , evn = bv16_byte ( inc_BV16 ( bv_DE ) ) ∧
        bvn , cvn = bv16_byte ( dec_BV16 ( bv_BC ) )
    THEN
        updateAddressMem ( bv_DE , bv_9HL0 ) ||
        rgs8 :=   rgs8 <+ { ( h0 ↦ hvn ) , ( l0 ↦ lvn ) ,
            ( d0 ↦ dvn ) , ( e0 ↦ evn ) ,
            ( b0 ↦ bvn ) , ( c0 ↦ cvn ) ,
            update_flag_reg( z_s , z_z , 0 , 0 , 0 , z_c ) }
         || r_ := update_refresh_reg(r_) ||
        IF   is_zero16USHORT ( bv16_ushort ( dec_BV16(bv_BC)   )   ) = 0
        THEN pc := instruction_next ( pc )   END
    END
END;

LDD =
PRE bv16_ushort(bv_DE) ∈ DATA_R_ADR     THEN
    ANY   hvn , lvn , dvn , evn , bvn , cvn
    WHERE
        hvn , lvn = bv16_byte ( dec_BV16 ( bv_HL ) ) ∧
        dvn , evn = bv16_byte ( dec_BV16 ( bv_DE ) ) ∧
        bvn , cvn = bv16_byte ( dec_BV16 ( bv_BC ) )
    THEN
        updateAddressMem ( bv_DE , bv_9HL0 ) ||
        rgs8 :=   rgs8 <+ { ( h0 ↦ hvn ) , ( l0 ↦ lvn ) ,
            ( d0 ↦ dvn ) , ( e0 ↦ evn ) ,
            ( b0 ↦ bvn ) , ( c0 ↦ cvn ) ,
            update_flag_reg( z_s , z_z , 0 ,
            bit_not ( is_zero16USHORT ( bv16_ushort ( dec_BV16(bv_BC)   )   ) ) , 0 , z_c ) } ||
```

$pc := instruction\_next\ (\ pc\ )\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

LDDR =
PRE $bv16\_ushort(bv\_DE) \in DATA\_R\_ADR$ **THEN**
ANY $hvn\ ,\ lvn\ ,\ dvn\ ,\ evn\ ,\ bvn\ ,\ cvn$
WHERE
$hvn\ ,\ lvn = bv16\_byte\ (\ dec\_BV16\ (\ bv\_HL\ )\ )\ \wedge$
$dvn\ ,\ evn = bv16\_byte\ (\ dec\_BV16\ (\ bv\_DE\ )\ )\ \wedge$
$bvn\ ,\ cvn = bv16\_byte\ (\ dec\_BV16\ (\ bv\_BC\ )\ )$
THEN
updateAddressMem $(\ bv\_DE\ ,\ bv\_9HL0\ )\ ||$
$rgs8 := rgs8 \leftarrow \{\ (\ h0 \mapsto hvn\ )\ ,\ (\ l0 \mapsto lvn\ )\ ,$
$(\ d0 \mapsto dvn\ )\ ,\ (\ e0 \mapsto evn\ )\ ,$
$(\ b0 \mapsto bvn\ )\ ,\ (\ c0 \mapsto cvn\ )\ ,$
$update\_flag\_reg(\ z\_s\ ,\ z\_z\ ,\ 0\ ,\ 0\ ,\ 0\ ,\ z\_c\ )\ \}$
$||\ r\_ := update\_refresh\_reg(r\_)\ ||$
IF $is\_zero16USHORT\ (\ bv16\_ushort\ (\ dec\_BV16(bv\_BC)\ )\ )\ = 0$
THEN $pc := instruction\_next\ (\ pc\ )$ **END**
END
END

ext_NMI =
PRE $bv16\_ushort(sp\_minus\_two) \in STACK\_R\_ADR \wedge bv16\_ushort(sp\_minus\_one) \in STACK\_R\_ADR$ **THEN**
ANY pc_low, pc_high **WHERE** $pc\_low \in BYTE \wedge pc\_high \in BYTE \wedge$
$bv16\_byte(ushort\_bv16(pc)) = (pc\_low,pc\_high) \wedge$
$sp\_minus\_two \in BV16 \wedge sp\_minus\_one \in BV16$
THEN
updateStack$(\{\ (sp\_minus\_two \mapsto pc\_low),(sp\_minus\_one \mapsto pc\_high\ )\ \})\ ||\ sp := sp\_minus\_two\ ||$
$pc := 102\ ||\ iff1 := 0\ ||\ iff2 := 0\ ||\ r\_ := update\_refresh\_reg(r\_)$
END
END;

ext_INT$(byte\_bus)$ =
PRE $iff1 = 1 \wedge byte\_bus \in 0\ ..\ 255 \wedge bv16\_ushort(sp\_minus\_one) \in STACK\_R\_ADR \wedge$
$bv16\_ushort(sp) \in STACK\_R\_ADR$
THEN
ANY pc_low, pc_high
WHERE $pc\_low \in BYTE \wedge pc\_high \in BYTE \wedge$
$bv16\_byte(ushort\_bv16(pc)) = (pc\_low,pc\_high)$
THEN
IF $im = (\ 0 \mapsto 0\ )$ **THEN**
IF $byte\_bus = 199 \vee byte\_bus = 207 \vee$
$byte\_bus = 215 \vee byte\_bus = 223 \vee$
$byte\_bus = 231 \vee byte\_bus = 239 \vee$
$byte\_bus = 247 \vee byte\_bus = 255$
THEN
$pc := byte\_bus - 199\ ||$
updateStack$(\ \{\ sp\_minus\_one \mapsto pc\_low,$
$sp\_minus\_two \mapsto pc\_high\ \}\ )\ ||$
$sp := sp\_minus\_two\ ||\ r\_ := update\_refresh\_reg(r\_)$
ELSE
skip

```
            END
        ELSIF  im =  ( 0 ↦ 1 ) THEN
            pc :=  56                          ||
            updateStack( { sp_minus_one ↦ pc_low,
                           sp_minus_two ↦ pc_high } ) ||
            sp := sp_minus_two ||    r_ := update_refresh_reg(r_)
        ELSIF    im = ( 1 ↦ 1 ) THEN
            pc := bv16_ushort(byte_bv16( i_ ,bitclear(rotateleft(uchar_byte(byte_bus)),0))) ||
            updateStack( {   sp_minus_one ↦ pc_low,
                           sp_minus_two ↦ pc_high } ) ||
            sp := sp_minus_two ||    r_ := update_refresh_reg(r_)
        END
    END
END;

ext_RESET =
BEGIN
    iff1:=0 || iff2:=0 || im:= (0 ↦ 0) || pc:=0 || i_ := uchar_byte(0) ||    r_ := uchar_byte(0) ||
    rgs8  := rgs8 <+ { (a0 ↦ uchar_byte(255) ), (f0 ↦ uchar_byte(255) ) } ||
    sp := byte_bv16(uchar_byte(255),uchar_byte(255))
END;

value ← io_read(aa) =
PRE aa ∈ id_reg_8 THEN
    value := rgs8(aa)
END;

io_write(aa,value) =
PRE aa ∈ id_reg_8 ∧ value ∈ BYTE THEN
    rgs8(aa) := value
END
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo