República Federativa do Brasil
Ministério do Desenvolvimento, Indústria e do Comércio Exterior
Instituto Nacional da Propriedade Industrial

(21) BR 10 2012 008476-7 A2

* B R 1 0 2 0 1 2 0 0 8 4 7 6 A 2 *

(22) Data de Depósito: 11/04/2012
(43) Data da Publicação: 26/11/2013
(RPI 2238)

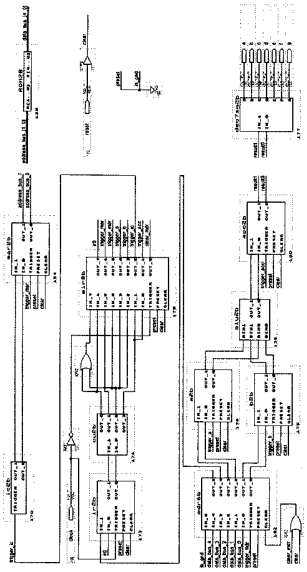
(51) Int.Cl.:
H03K 17/00
H03K 19/177

(54) Título: MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES

(73) Titular(es): UNIVERSIDADE ESTADUAL PAULISTA "Júlio de Mesquita Filho"

(72) Inventor(es): MAURÍCIO ARAÚJO DIAS

(57) Resumo: MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES. Patente de invenção de um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações que compreende uma arquitetura de computador implementada em FPGA (Field Programmable Gate Array).



MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES

CAMPO DA INVENÇÃO

A presente invenção descreve um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações. Mais especificamente compreende uma arquitetura de computador implementada em FPGA (*Field Programmable Gate Array*).

ANTECEDENTES DA INVENÇÃO

Se o presente da Ciência da Computação ainda é marcado pelo forte desenvolvimento de *software*, o seu futuro incluirá também um forte desenvolvimento de *hardware*, pois as tecnologias disponíveis no mercado proporcionarão o desenvolvimento de soluções computacionais implementadas fisicamente de maneira mais fácil, rápida, barata e, sobretudo, flexível.

Alguns anos atrás, as ferramentas implementadas em *software*, em geral, eram caras e possuíam código fechado. Hoje, os programas têm custo acessível ao usuário comum, quando não são gratuitos, e a maioria pode ser adquirida facilmente pela Internet. Além disso, muitas dessas ferramentas são oferecidas com código aberto, o que possibilita ao cientista da computação fazer atualizações e otimizações, sem ou com poucas restrições. Quando isso ocorre, qualquer tipo de *software*, como editores gráficos, jogos, antivírus, compiladores e até mesmo sistemas operacionais, entre outros, pode ser reconfigurado pelo próprio cientista da computação, de acordo com as suas necessidades. Esses códigos abertos e de baixo custo potencializam a capacidade de criação e investigação dos cientistas da computação, além de estimular pesquisas envolvendo *software*.

Por outro lado, a realidade do *hardware* tem permanecido a mesma há anos: implementações caras e documentações com

acesso restrito. Consequentemente, as pesquisas científicas em Ciência da Computação envolvendo *hardware* são realizadas, predominantemente, de forma teórica, deixando a prática mais para a Engenharia da Computação. A razão para isso é a alta complexidade e o alto custo para o desenvolvimento de *hardware* através do emprego de métodos tradicionais, como implementações de circuitos diretamente impressos em silício, por exemplo.

A abordagem exclusivamente teórica da Ciência da Computação permite ao cientista da computação adquirir apenas uma visão geral e superficial do conhecimento apresentado nessa área. Neste caso, não há qualquer possibilidade de interação entre o cientista da computação e o *hardware*, pois todo conhecimento adquirido é proveniente de informações teóricas.

Para que o cientista da computação adquira uma visão mais aprofundada sobre os temas abordados dentro dessa área e seja capaz de criar produtos tecnológicos otimizados, é necessário que ele tenha maior contato com diferentes implementações em *hardware*, o que parecia ser inviável até então.

Porém, assim como historicamente aconteceu com o *software*, a realidade do *hardware* promete mudar em pouco tempo. A tendência futura, já no curto prazo, é a de arquiteturas de computadores cada vez mais acessíveis ao mercado comum ou até mesmo gratuitas. As documentações e descrições de comportamento poderão ser disponibilizadas a baixo custo ou gratuitamente para serem adquiridas pela Internet. Isso significa que será possível fazer *download* de *hardware* virtual, atualizado e otimizado graças a uma tecnologia atual que suporta reconfigurações. Dito de outra forma, qualquer pessoa, em qualquer lugar do mundo, poderá adquirir, por exemplo, uma nova arquitetura de computador de baixo custo através da Internet.

Cientistas da computação poderão facilmente otimizar arquiteturas de computadores segundo as suas necessidades. Além disso, eles poderão disponibilizá-las para a comunidade científica e para o mercado tecnológico através de *upload* na Internet.

Dessa forma, esse *hardware* reconfigurável contribuirá para aumentar o poder de criação dos cientistas da computação, além de estimular pesquisas envolvendo *hardware*.

Dessa forma, é possível empregar ferramentas implementadas em *software* com o objetivo de levar um conhecimento um pouco mais profundo ao cientista da computação e permitir as reconfigurações de *hardware*. Convencionalmente, esse *software* emula um ambiente de desenvolvimento de *hardware*, permitindo que o cientista da computação chegue um pouco mais próximo daquilo que é a implementação física de uma ou mais soluções computacionais. Tais ferramentas também possibilitam validar o funcionamento dos dispositivos criados através de simuladores virtuais. Entretanto, um cientista da computação somente vai alcançar um nível de conhecimento com profundidade ideal para dominar o *hardware*, propor e implementar otimizações em circuitos, se ele tiver um contato direto e prolongado com o desenvolvimento de distintas soluções computacionais em *hardware*. Isso significa que a pesquisa ideal envolvendo circuitos digitais, microcontroladores, microprocessadores e arquiteturas de computadores é aquela em que os cientistas da computação não somente desenvolvem e analisam o *hardware*, como também o reconfigurem, o sintetizem e o simulem em sua forma física, o maior número de vezes possível, de forma a refinar a sua otimização.

As condições para a implementação desse *hardware* reconfigurável são oportunizadas por uma plataforma digital baseada

em uma tecnologia atual e bastante conhecida – a plataforma FPGA (*Field Programmable Gate Array*).

O FPGA é uma plataforma que se apresenta na forma de um chip, dita plataforma constituída por um arranjo eletrônico que lhe dá capacidade de trabalhar como qualquer circuito digital. Dito de outra forma, é possível "programar" uma *FPGA* através da descrição comportamental de um *hardware* para que ela passe a se comportar como qualquer circuito digital básico, microcontrolador, microprocessador ou mesmo como uma arquitetura de computador complexa. Essa "programação" é feita com base na descrição do comportamento do *hardware* através do uso de uma linguagem de descrição, como a *VHDL – VLSI Hardware Description Language*, por exemplo, ou em esquema elétrico e, em seguida, para ambos os casos, a descrição tem que ser sintetizada e descarregada na *FPGA*. O número de vezes que uma *FPGA* pode ser "reprogramada" é teoricamente ilimitado. Isso a classifica como uma plataforma ideal para sustentar atualizações e otimizações de *hardware*, através da reconfiguração do mesmo. O que dá essa capacidade a uma *FPGA* é o fato de ela ser dotada de elementos lógicos que podem assumir o comportamento de portas lógicas (os blocos básicos fundamentais usados na construção de qualquer circuito digital, inclusive processadores). Assim, circuitos digitais básicos, microcontroladores, microprocessadores e arquiteturas de computadores podem ser implementados fisicamente de forma fácil, rápida, barata e, sobretudo, flexível, graças às *FPGAs*.

O uso de *FPGAs* permitirá, ao cientista da computação, adquirir notável nível de conhecimento e, consequentemente, desenvolver produtos tecnológicos otimizados ao trabalhar com temas de pesquisa que envolvem circuitos digitais, microcontroladores,

microprocessadores e arquiteturas de computadores, envolvendo pouca complexidade de projeto e a um custo relativamente baixo. Isso porque a implantação em *FPGAs* possibilitará que o cientista da computação reconfigure, sintetize e simule seus trabalhos em uma plataforma física real, quantas vezes forem necessárias a fim de otimizar os circuitos em desenvolvimento.

O estado da técnica no que se refere à tecnologia de processadores apresenta resultados de pesquisas que poderiam ter sido desenvolvidas mais rápida e facilmente se utilizado como ponto de partida o método para arquitetura de computador reconfigurável e sujeita a constantes otimizações, implementada em FPGA, conforme segue:

M. A. Iqbal et. Al (M. A. Iqbal, U. S. Awan, and S. A. Khan, "Reconfigurable Computing Technology Used for Modern Scientific Applications", In IEEE 2nd International Conforence on Education Technology and Computer (ICETC), 2010) realiza uma análise comparativa entre os métodos tradicionais usados para o desenvolvimento de circuitos digitais e a abordagem reconfigurável.

L. Yan et al. (L. Yan, B. Wu, Y. Wen, Shaobin Zhang, and Tianzhou Chen, "A reconfigurable processor architecture combining multi-core and reconfigurable processing unit", In 10th IEEE International Conference on Computer and Information Technology (CIT 2010), 2010) propõem uma arquitetura de processador reconfigurável, a qual combina múltiplos núcleos com unidades de processamento re-configuráveis.

A. Dollas (Dollas, "Reconfigurable Architectures for Bioinformatics Applications", In IEEE Annual Symposium on VLSI, 2010) descreve uma arquitetura re-programável desenvolvida para executar algoritmos da bioinformática, tais como comparações entre cadeias de

DNA, filogenética, dentre outros.

W. Q. Lu et al. (W. Q. Lu, S. Zhao, X. F. Zhou, J. Y. Ren, and G. E. Sobelman, "Instructionless Processor Architecture Using Dynamically Reconfigurable Logic", In IET Computers & Digital Techniques, 2010) apresentam uma arquitetura reconfigurável, que é uma plataforma flexível e eficiente para processamento digital de sinais.

R. Kielbik et al. (R. Kielbik, G. Jablonski, B. Swiercz, and P. Amrozik, "Instructionless Processor Architecture Using Dynamically Reconfigurable Logic", In 17th International Conference "Mixed Design of Integrated Circuits and Systems (MIXDES), 2010) propõem uma arquitetura de processador com conjunto reduzido de instruções, que usa lógica dinamicamente reconfigurável e executa todas as instruções em paralelo.

K. A. Fawaz et al. (K. A. Fawaz, T. Arslan, S. Khawam, M. Muir, I. Nousias, I. Lindsay, and A. Erdogan, "A Dynamically Reconfigurable Asynchronous Processor", In IEEE 8th Symposium on Application Specific Processors (SASP), 2010) apresentam um processador assíncrono de alto desempenho dinamicamente reconfigurável, que pode ser programado e que consome pouca energia.

P. Patel and M. Moallem (P. Patel and M. Moallem, "Reconfigurable system for real-time embedded control applications", In IET Computers & Digital Techniques, 2010) discutem o desenvolvimento de controladores embarcados sobre um sistema multiprocessador reconfigurável, usando FPGA.

D. Göhringer et al. (D. Göhringer, M. Hübner, M. Benz, and J. Becker, "A Design Methodology for Application Partitioning and Architecture Development of Reconfigurable Multiprocessor Systems-on-Chip", In 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, 2010) descrevem uma

metodologia de projeto para o desenvolvimento de arquiteturas de sistemas multiprocessadores re-configuráveis, a qual ajuda a esconder a complexidade do hardware e garante alternativas de projeto ao desenvolvedor.

D. Göhringer and J. Becker (D. Göhringer and J. Becker, "High Performance Reconfigurable Multi-Processor-Based Computing on FPGAs", In IEEE, 2010) apresentam uma nova abordagem para computação baseada em multiprocessadores re-configuráveis de alto desempenho sobre FPGAs.

P. J. Pingree (P. J. Pingree, "Advancing NASA's On-Board Processing Capabilities with Reconfigurable FPGA Technologies: Opportunities & Implications", Keynote Talk, IEEE, 2010) comenta sobre o impacto positivo do uso da computação reconfigurável em futuras missões da NASA.

SUMÁRIO

É característica da invenção um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações caracterizado pela facilidade, rapidez e baixo custo.

É característica da invenção um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações, implementada em FPGA que serve de modelo base para o desenvolvimento de computadores para a pesquisa científica e para o mercado.

É característica da invenção um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações, implementada em FPGA implementada em uma plataforma re-programável, permitindo a constante re-configuração da arquitetura.

É característica da invenção um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações,

implementada em FPGA que permite a aquisição exclusivamente pela internet, o que minimiza ainda mais o custo do produto.

BREVE DESCRIÇÃO DAS FIGURAS

A figura 1 apresenta o esquema elétrico da primeira reconfiguração.

A figura 2 apresenta a Unidade de controle (*CU*) *hardwired* da primeira reconfiguração.

A figura 3 apresenta o Contador de instruções (*IC*) da primeira reconfiguração.

A figura 4 apresenta a memória *ROM* da primeira reconfiguração

A figura 5 apresenta a unidade aritmética e lógica (*ALU* - circuito somador) da primeira reconfiguração.

A figura 6 apresenta a unidade básica dos registradores da primeira reconfiguração.

A figura 7 apresenta o esquema elétrico da segunda reconfiguração.

A figura 8 apresenta a unidade de controle (*CU*) microprogramada da segunda reconfiguração.

A figura 9 apresenta a ALU da segunda reconfiguração.

A figura 10 apresenta o esquema elétrico da terceira reconfiguração.

A figura 11 apresenta o registrador de dados da memória (*MDR*) da terceira reconfiguração.

A figura 12 apresenta o esquema elétrico da quarta reconfiguração.

A figura 13 apresenta o esquema elétrico da quinta reconfiguração.

A figura 14 apresenta a unidade de controle (*CU*) microprogramada da quinta reconfiguração.

A figura 15 apresenta o esquema elétrico da sexta reconfiguração.

A figura 16 apresenta parte do *MDR* da sexta reconfiguração.

A figura 17 apresenta o esquema elétrico da sétima reconfiguração.

A figura 18 apresenta o esquema elétrico da oitava reconfiguração.

A figura 19 apresenta o esquema elétrico da nona reconfiguração.

DESCRIÇÃO DETALHADA DA INVENÇÃO

A arquitetura de computador compreende convencionalmente uma unidade de controle (*CU*) *hardwired,* a qual trabalha em conjunto com um registrador de instrução (*IR*) e com um registrador de micro-instrução (*MIR*) para controlar as demais unidades funcionais da mesma arquitetura.

A arquitetura de computador possui, ainda, um contador de instruções (*IC*) e um registrador de endereço da memória (*MAR*), os quais, juntos, permitem referenciar cada uma das quatro palavras de memória (de cinco bits cada uma) presentes em uma memória *ROM*, (esta última está descrita em *VHDL*, conforme apresentado na figura 4).

Um barramento de endereços de dois bits de largura liga o registrador de endereço da memória (*MAR)* à memória *ROM*. A arquitetura de computador também conta com um registrador de dados da memória (*MDR*), usado para receber as informações provenientes da memória *ROM*, através de um barramento de dados de dois bits de largura. Há, ainda, dois registradores auxiliares (*A* e *B*) e uma unidade aritmética e lógica (*ALU*). A unidade aritmética e lógica (*ALU)*, conforme mostrada pela Figura 5, é implementada como um

circuito somador, o qual permite a realização de somas dentro da arquitetura. Dando seqüência, um acumulador (*ACC*) é usado para guardar os resultados dessas operações. O acumulador (*ACC)* e demais registradores são representados pela Figura 6.

O resultado registrado pelo acumulador (*ACC)* pode ser visualizado através de um *display* de sete segmentos, graças à presença de um decodificador especial (dec7sg) presente na arquitetura. Existe, também, uma entrada para receber um sinal *power* ou *reset*, necessário para iniciar (ou reiniciar) a atividade da arquitetura e uma entrada para receber o sinal de *clock*.

Esta e as demais arquiteturas trabalham com uma freqüência experimental de *clock* igual a 10MHz, apesar dessa não ser a freqüência máxima em que as arquiteturas possam trabalhar.

O método para arquitetura de computador reconfigurável e sujeita a constantes otimizações, objeto da presente invenção, compreende um ciclo de busca e um ciclo de execução.

Conforme apresentado na figura 4, cada instrução do programa é formada por cinco bits. O mais significativo deles, ou seja, o bit mais à esquerda, representa o código da operação, cujo valor "1" identifica uma operação aritmética de soma. Os dois bits do meio representam o primeiro operando dessa soma. Os dois bits menos significativos representam o segundo operando da mesma soma.

Dessa forma, por exemplo, para programar a soma de dois números decimais (1 + 2), é preciso escrever na memória *ROM*, a instrução: 1 01 10.

A arquitetura proposta na presente invenção pode ser reiniciada sempre que o botão "reset" for pressionado. Nesta situação, todos os registradores passam a armazenar o valor zero. Após isso, o primeiro evento que efetivamente ocorre nessa arquitetura é a passagem de

eletricidade pelo *clock*. O *clock* é composto por uma lâmina de cristal de quartzo, a qual vibra ininterruptamente enquanto estiver sendo estimulada por eletricidade. Essa vibração faz com que a corrente elétrica, presente em um dos terminais ligados ao cristal apresente um comportamento típico de onda quadrática. Dito de outra forma, o cristal gera variações de sinais elétricos nos estados "0" e "1". Essas variações são usadas para sincronizar os eventos que ocorrem dentro da arquitetura.

O ciclo de busca da arquitetura é dividido em duas fases: a primeira fase ocorre quando o registrador de instrução (*IR)* está armazenando o valor "00"; a segunda fase quando o registrador de instrução (*IR)* armazena "01".

Na primeira fase, quando o *clock* está em "0", o conteúdo do registrador de instrução *(IR)*, que nesse momento é "00", chega até a unidade de controle (*CU)*. Esse valor, que pode representar tanto o código de uma instrução como de uma micro-instrução, é decodificado pela unidade de controle (*CU)*. Neste caso, "00" representa a busca pela instrução de um programa armazenado na memória. O resultado dessa decodificação é a geração de uma micro-instrução que fica à disposição do registrador de micro-instrução (*MIR)*.

O registrador de micro-instrução (*MIR)* recebe e armazena a micro-instrução proveniente da unidade de controle (*CU)* porque a sua entrada *trigger* está recebendo o valor "1" nesse instante de tempo. Esse valor "1" é resultante da inversão do sinal do *clock* por uma porta lógica do tipo NOT. A micro-instrução armazenada no registrador de micro-instrução (*MIR)* representa um conjunto de sinais de controle que, a partir desse registrador, servem para controlar uma ou mais unidades funcionais da arquitetura ao mesmo tempo. Assim, nas saídas do registrador de micro-instrução (*MIR)*, são disponibilizados

os sinais de controle *trigger_mar* = 1 e *ir_0* = 1, nesse momento. Os demais sinais de controle apresentam valores iguais a "0". O sinal de controle chamado *trigger_mar*, cujo valor é "1", é responsável por permitir que o registrador de endereço da memória (*MAR)* armazene o valor "00" proveniente do contador de instruções (*IC)*. O contador de instruções (IC) gera e armazena o endereço de memória da próxima instrução que deverá ser executada pelo processador. Dessa forma, por intermédio do registrador de endereço da memória (*MAR)*, esse endereço sai do processador e chega até a memória, onde é decodificado. Essa decodificação permite que o conteúdo da célula de memória, cujo endereço é "00", seja disponibilizado, a partir dessa posição de memória, para o registrador de dados da memória (*MDR)*, interno ao processador, dito registrador de dados da memória (*MDR)* que nesta fase não pode armazenar esse conteúdo, pois a sua entrada *trigger* está com valor "0".

Na segunda fase, o *clock* muda de valor automaticamente, passando de "0" para "1". Quando isso acontece, o registrador de instrução (*IR)* recebe "1" na sua entrada *trigger*, o que o habilita a armazenar o conteúdo das linhas *ir_1* e *ir_0* que, juntas, apresentam o valor "01", o qual é usado para indicar o código da próxima micro-instrução a ser fornecida pela unidade de controle (*CU)*.

Dessa forma, a partir do registrador de instrução (*IR)*, o valor "01" chega até a unidade de controle (*CU)*, onde é decodificado. O resultado dessa decodificação é a geração de uma micro-instrução que fica à disposição do registrador de micro-instrução (*MIR), que* ainda não pode armazenar a nova micro-instrução, pois a sua entrada *trigger* está com valor "0".

Em um instante seguinte, o valor do *clock* muda de "1" para "0". Consequentemente, a entrada *trigger* do registrador de micro-

instrução (*MIR*) passa a receber o valor "1". Assim, o registrador de micro-instrução (*MIR*) armazena a micro-instrução e fornece um novo conjunto de sinais de controle nas suas saídas. Nesse momento, o único sinal de controle com valor igual a "1", nas saídas do registrador de micro-instrução (*MIR*) é o *trigger_mdr*. Este sinal permite ao registrador de dados da memória (*MDR*) armazenar a instrução proveniente da memória.

A saída *out4* do registrador de dados da memória (*MDR*) está diretamente ligada à entrada *ir_1* do registrador de instrução (*IR*), por onde o código da instrução recebida chega ao registrador de instrução (*IR*), dito código cujo valor "1" representa uma instrução de soma, a qual é executada pelo processador no ciclo de execução.

O ciclo de execução apresenta duas fases, sendo uma primeira fase que ocorre quando o registrador de instrução (*IR*) está armazenando o valor "10"; e a segunda fase quando o registrador de instrução (*IR*) armazena "11".

Na primeira fase, o *clock* muda de valor automaticamente, passando de "0" para "1". Quando isso acontece, o registrador de instrução (*IR*) recebe "1" na sua entrada *trigger*, o que o habilita a armazenar o conteúdo das linhas *ir_1* e *ir_0, que, juntas,* agora apresentam o valor "10", o qual é usado para indicar o código da próxima micro-instrução a ser fornecida pela unidade de controle (*CU*). Dessa forma, a partir do registrador de instrução (*IR*), o valor "10" chega até a unidade de controle (*CU*), onde é decodificado. O resultado dessa decodificação é a geração de uma micro-instrução que fica à disposição do registrador de micro-instrução (*MIR*) que ainda não pode armazenar a nova micro-instrução pois a sua entrada *trigger* está com valor "0".

No instante seguinte, o valor do *clock* muda de "1" para "0".

Consequentemente, a entrada *trigger* do registrador de micro-instrução (*MIR)* passa a receber o valor "1". Assim, o registrador de micro-instrução (MIR) consegue armazenar a micro-instrução e fornecer um novo conjunto de sinais de controle nas suas saídas. Dessa forma, nas saídas do registrador de micro-instrução (MIR) tem-se os sinais de controle *trigger_a* = 1, *trigger_b* = 1, *trigger_ic* = 1 e *ir_0* = 1, nesse momento. Os demais sinais de controle apresentam valores iguais a "0". O sinal de controle chamado *trigger_ic*, cujo valor é "1", é responsável por incrementar o valor do contador de instruções (*IC)*. O contador de instruções (IC) passa, então, a apontar para o endereço da próxima instrução que somente será buscada na memória quando um novo ciclo de busca for iniciado, ou seja, não agora, só depois do termino completo do ciclo de execução.

O sinal de controle chamado *trigger_a*, cujo valor é "1", é responsável por permitir que o registrador *A* armazene o primeiro operando da instrução. O sinal de controle chamado *trigger_b*, cujo valor é "1", é responsável por permitir que o registrador *B* armazene o segundo operando da instrução.

A partir dos registradores *A* e *B*, ambos os operandos são passados para a unidade aritmética e lógica (*ALU)*, onde são somados. O resultado dessa soma fica à disposição do registrador *ACC*, nas saídas da unidade aritmética e lógica (*ALU)*. Entretanto, o registrador *ACC* ainda não pode armazenar o resultado da soma, pois a sua entrada *trigger* está com valor "0".

Na segunda fase, o *clock* muda de valor automaticamente., passando de "0" para "1". Quando isso acontece, o registrador de instrução (*IR)* recebe "1" na sua entrada *trigger*, o que o habilita a armazenar o conteúdo das linhas *ir_1* e *ir_0 que* juntas, agora apresentam o valor "11", o qual é usado para indicar o código da

próxima micro-instrução a ser fornecida pela unidade de controle (*CU*). Dessa forma, a partir do registrador de instrução (*IR*), o valor "11" chega até a unidade de controle (*CU*), onde é decodificado. O resultado dessa decodificação é a geração de uma micro-instrução, a qual ficará à disposição do registrador de micro-instrução (*MIR*). Entretanto, este registrador ainda não pode armazenar a nova microinstrução, pois a sua entrada *trigger* está com valor "0".

Em um instante seguinte, o valor do *clock* muda de "1" para "0". Consequentemente, a entrada *trigger* do registrador de micro-instrução (*MIR*) passa a receber o valor "1", dito registrador que consegue armazenar a micro-instrução e fornecer um novo conjunto de sinais de controle nas suas saídas. Nesse momento, o único sinal de controle com valor igual a "1" nas saídas do *MIR* é o *trigger_acc*. Este sinal permite ao registrador *ACC* armazenar o resultado da soma disponibilizado pelas saídas da unidade aritmética e lógica (*ALU*).

A reconfiguração da primeira arquitetura, conforme apresentado na figura 7, funciona exatamente como a primeira arquitetura e ambas implementam as mesmas unidades funcionais.

Nesta segunda reconfiguração, enquanto a unidade de controle (*CU*) da primeira reconfiguração é *hardwired*, a segunda reconfiguração implementa uma unidade de controle (*CU*) microprogramada, descrita em VHDL (conforme apresentada pela Figura 8), porém mantendo o mesmo conjunto reduzido de instruções anterior.

A unidade aritmética e lógica (*ALU*) desta segunda arquitetura, mostrada pela Figura 9, vai mais além de apenas um circuito somador, dita unidade aritmética e lógica (ALU) que implementa também os circuitos que permitem a realização das operações lógicas AND, OR, NOT e XOR. A seleção de uma dentre essas cinco

operações possíveis de serem realizadas pela unidade aritmética e lógica (*ALU*) é feita com o auxílio de um circuito multiplexador presente em seu interior. Entretanto, em um primeiro momento, apenas a operação de soma será selecionada.

A última diferença entre as duas arquiteturas é encontrada no decodificador do "display" de sete segmentos. Ele foi projetado originalmente para representar decimais de zero a três, mas nesta nova arquitetura ele foi repensado para poder mostrar um intervalo maior de números decimais, indo do 0 ao 9. O mesmo decodificador conta, agora, com entradas para quatro bits, já prevendo uma futura expansão do circuito, embora apenas duas delas sejam efetivamente usadas por enquanto.

A terceira reconfiguração, conforme mostrado na figura 10, é resultante do processo de reconfiguração da segunda reconfiguração. Nesta terceira reconfiguração é mantida a operacionalidade e as unidades funcionais da segunda reconfiguração, sendo acrescidas algumas características, tais como a reconfiguração do barramento de dados e a introdução do terceiro estado (o estado de alta impedância, representado pela letra Z). O barramento de dados foi transformado em um barramento bidirecional, ou seja, o barramento passou a suportar dados que vão e que voltam pelo mesmo caminho. O fluxo de dados nesse barramento é, a partir de agora, controlado através de um elemento chamado *tri-state*. A introdução desse elemento possibilita que várias unidades funcionais da arquitetura possam compartilhar um barramento (o de dados, neste caso). Enquanto uma dessas unidades estiver efetivamente colocando dados no barramento, as demais obrigatoriamente deverão ser mantidas em um estado de alta impedância, isto é, estarão temporariamente suspensas, para que não ocorra colisão entre os dados.

Nesta arquitetura, o código de instrução passa a ser representado por dois bits, ao contrário de um bit das arquiteturas anteriores. Desses dois bits, que também estão presentes no barramento de controle (como ir1 e ir0), o mais significativo deles também será controlado por elementos do tipo *tri-state*.

O registrador de micro-instrução *(MIR)*, que até então era um registrador de oito bits, passa a ser um registrador de dezesseis bits. Essa reconfiguração do registrador de micro-instrução *(MIR)* visa deixá-lo preparado para suportar uma futura expansão da arquitetura, embora, neste momento, apenas a metade da sua capacidade continue sendo explorada. Ele também vai incorporar um dos dois elementos *tri-state* que vão controlar o bit mais significativo do código de instrução.

O registrador de dados da memória *(MDR)*, apresentado pela Figura 11, é completamente reconfigurado para trabalhar com o barramento de dados bidirecional. Isso quer dizer que vários elementos do tipo *tri-state* foram implementados em seu esquema elétrico de modo que ele possa interagir com o barramento de dados. Há também um *tri-state* adicional que é usado para controlar o bit mais significativo do código de instrução.

E por fim, a unidade de controle *(CU)* começa a ter o seu código VHDL documentado, juntamente com a sua descrição. Isso também vai ocorrer em todas as arquiteturas posteriores a esta.

A quarta reconfiguração, conforme apresentado na Figura 12, é a própria terceira reconfiguração reconfigurada para trabalhar com quatro bits. Até este momento, todas as arquiteturas anteriores trabalhavam com apenas dois bits. A quarta reconfiguração implementa unidades funcionais e trabalha da mesma forma que a terceira reconfiguração, porém com a diferença que ela passa a ser

uma arquitetura de quatro bits. Os seus barramentos de dados e de endereço também possuem quatro bits de largura. Nesta arquitetura, a memória *ROM* foi reconfigurada de modo a armazenar dezesseis palavras de memória, de cinco bits cada uma. Apesar de ter sido reconfigurada de modo a poder trabalhar com quatro bits, ela é validada com a mesma operação sobre dois bits realizada pelas arquiteturas anteriores. Em outras palavras, ela trabalha com apenas metade do seu potencial.

A quinta reconfiguração, conforme apresentado na figura 13, preserva o mesmo conjunto de funcionalidades da quarta reconfiguração, com acréscimo de algumas características, tal como a possibilidade de realizar as operações lógicas AND, OR, NOT e XOR, além da soma. Um circuito multiplexador presente no interior da unidade aritmética e lógica *(ALU)* possibilita a seleção de uma dentre essas cinco operações. Essa unidade funcional comporta futuras implementações de circuitos aritméticos.

Consequentemente, a unidade de controle *(CU)* dessa arquitetura, apresentada pela Figura 14, implementa o micro-código necessário para controlar a execução de cada uma das operações oferecidas pela unidade aritmética e lógica *(ALU)*, além do micro-código usado para carregar os registradores *A* e *B* com dados provenientes da memória. Esses dados representam os valores que serão usados nas operações da unidade aritmética e lógica *(ALU)*.

Em decorrência das alterações realizadas na unidade de controle *(CU)*, o registrador de micro-instrução *(MIR)* passa a armazenar novas microinstruções.

Nesta arquitetura, a memória *ROM* foi reconfigurada de modo a poder armazenar dezesseis palavras de memória, de quatro bits cada uma.

Como cada dado proveniente da memória agora é composto por apenas quatro bits, o registrador de dados da memória (*MDR)* sofreu uma redução para trabalhar apenas com esses quatro bits.

A diferença mais significativa dessa arquitetura, em relação às anteriores, é a presença de códigos de instruções realmente compostos por quatro bits. Em outras palavras, a quinta reconfiguração tem capacidade para implementar dezesseis instruções básicas, em teoria. Na prática, implementa sete instruções complexas.

Na quinta reconfiguração, existem oito elementos *tri-state* que vão controlar os bits do código de micro-instrução: quatro implementados nas saídas do registrador de dados da memória (*MDR)* e outros quatro dentro do registrador de micro-instrução (*MIR).*

Na sexta reconfiguração, a unidade de controle (*CU)* trabalha com micro-instruções de trinta e dois bits. Isso permite ampliar o controle sobre as unidades funcionais da própria arquitetura. Consequentemente, o registrador de micro-instrução (*MIR)* passa a ser um registrador de trinta e dois bits. Além de manter o conjunto de instruções anterior, a micromemória incorporou uma nova instrução para carregar o registrador *ACC* diretamente com dados provenientes da memória.

Uma mudança significativa apresentada por essa arquitetura é a criação de um barramento interno bidirecional. Esse barramento interliga os registradores registrador de dados da memória (*MDR),* registradores *A, B* e *ACC,* além da unidade aritmética e lógica *(ALU).* Todas essas unidades funcionais precisam se ligar a esse barramento através de elementos *tri-state,* para que os seus dados não se choquem no próprio barramento. As quatro referidas unidades funcionais foram reconfiguradas de forma a incorporar os elementos

tri-state. O fluxo de dados do barramento interno é controlado pela unidade de controle (*CU)* através do envio de sinais de controle para esses elementos *tri-state*.

O registrador de dados da memória (*MDR)*, mostrado parcialmente pela Figura 16, foi modificado de modo a ter saídas exclusivas para o código de instrução. Os elementos *tri-state* que controlam essas saídas foram incorporados por esse registrador.

Nessa reconfiguração, a unidade aritmética e lógica *(ALU)* passou a ser realimentada pelo registrador *ACC*, o que proporciona a execução repetida e contínua de operações aritméticas e lógicas sobre um mesmo dado.

Na sétima reconfiguração, conforme apresentado na Figura 17, todos os resultados são mostrados através de dois *displays* de sete segmentos, graças à reconfiguração do decodificador do *display*.

A mudança marcante apresentada pela sétima reconfiguração é a implementação de uma memória de armazenamento temporário do tipo *RAM* e uma memória de armazenamento permanente do tipo *Flash*. Ambas trabalham com quatro bits de entrada e quatro de saída, podem armazenar dezesseis palavras de memória cada uma e estão ligadas ao barramento de dados através de elementos *tri-state*, controlados pela unidade de controle (*CU)*.

A presença da memória *RAM* nesta arquitetura dá condições para a implementação de uma nova instrução na micro-memória. Essa instrução é responsável por armazenar dados na memória *RAM*.

A oitava reconfiguração, conforme apresentado na Figura 18, é uma arquitetura de oito bits. A largura dos seus barramentos de dados e de endereço também passou a ser de oito bits.

A nona reconfiguração, conforme apresentado na Figura 19, trabalha com oito bits, permitindo que trabalhe com até sessenta

instruções básicas, porém ela implementa um número menor de instruções complexas.

Adicionalmente, a nona reconfiguração implementa três contadores distintos, chamados contador de instruções (*IC*), *IX* e *SP*, os quais permitirão que esta nona arquitetura passe a trabalhar com outros modos de endereçamento além do modo imediato usado até agora. Esses três contadores enviam endereços ao *MAR* através de um barramento multiplexado.

Na nona reconfiguração, a memória *ROM* foi reconfigurada de forma a poder guardar sessenta e quatro palavras de memória, de oito bits cada uma.

A unidade de controle (*CU*) passou a incorporar o registrador de instrução *(IR)* e o registrador de micro-instrução (*MIR*), além da já presente micro-memória microprogramada. Esta última gera micro-instruções de sessenta e quatro bits para controlar o restante da arquitetura. Isso quer dizer que, internamente, a nona reconfiguração pode ser controlada por até sessenta e quatro sinais de controle distintos. Em decorrência desse aumento de bits na micro-memória, o registrador de micro-instrução (*MIR*) também passou a trabalhar com sessenta e quatro bits e o barramento de controle passou a ter sessenta e quatro bits de largura. Outro detalhe da unidade de controle (*CU*) é que ela recebe e trata um sinal de interrupção proveniente de um timer. Esse timer pode ser programado para interromper o processamento corrente em um dado instante de tempo pré-determinado.

Outra novidade é a implementação de um registrador chamado *PSW*, o qual armazena o estado corrente de um programa em execução.

A nona reconfiguração apresenta os seus resultados através de

dois *displays* de sete segmentos, por isso o decodificador do *display* foi ampliado para suportá-los.

REIVINDICAÇÕES

1. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES que compreende uma unidade de controle (*CU*) *hardwired* que trabalha em conjunto com um registrador de instrução (*IR*) e com um registrador de micro-instrução (*MIR*) para controlar as demais unidades funcionais da mesma arquitetura, um contador de instruções (*IC*) e um registrador de endereço da memória (*MAR*), os quais, juntos, permitem referenciar cada uma das quatro palavras de memória (de cinco bits cada uma) presentes em uma memória *ROM*, um barramento de endereços de dois bits de largura que liga o registrador de endereço da memória (*MAR)* à memória *ROM,* um registrador de dados da memória (*MDR*) para receber as informações provenientes da memória *ROM* através de um barramento de dados de dois bits de largura, dois registradores auxiliares (*A* e *B*) e uma unidade aritmética e lógica (*ALU*) implementada como um circuito somador, um acumulador (*ACC*) para guardar os resultados dessas operações da unidade aritmética e lógia (ALU) e um display que permite visualizar o resultado registrado pelo acumulador (*ACC)* através de um decodificador especial (dec7sg) caracterizado por compreender as etapas de:

1. um ciclo de busca que apresenta:

a.1) uma primeira fase em que o *clock* está em "0" e o conteúdo do registrador de instrução *(IR)* é "00", chega até a unidade de controle (*CU)* para ser decodificado, cujo resultado é a geração de uma micro-instrução que fica à disposição do registrador de micro-instrução (*MIR) que* recebe e armazena a micro-instrução proveniente da unidade de controle (*CU)*, em função da sua

entrada *trigger* estar recebendo o valor "1" nesse instante de tempo, dito valor "1" resultante da inversão do sinal do *clock* por uma porta lógica do tipo NOT, sendo disponibilizadas nas saídas do registrador de micro-instrução (*MIR*) sinais de controle *trigger_mar* = 1 e *ir_0* = 1, e os demais sinais de controle apresentando valores iguais a "0", de forma que o contador de instruções (IC) gera e armazena o endereço de memória da próxima instrução que deverá ser executada pelo processador e, por intermédio do registrador de endereço da memória (*MAR*), esse endereço sai do processador e chega até a memória, onde é decodificado, permitindo que o conteúdo da célula de memória, cujo endereço é "00", seja disponibilizado, a partir dessa posição de memória, para o registrador de dados da memória (*MDR*), interno ao processador, dito registrador de dados da memória (*MDR*) que nesta fase não pode armazenar esse conteúdo, pois a sua entrada *trigger* está com valor "0";

a.2) uma segunda fase onde o *clock* muda de valor automaticamente, passando de "0" para "1", de forma que o registrador de instrução (*IR*) recebe "1" na sua entrada *trigger*, habilitando-o a armazenar o conteúdo das linhas *ir_1* e *ir_0* que, juntas, apresentam o valor "01", o qual é usado para indicar o código da próxima micro-instrução a ser fornecida pela unidade de controle (*CU*) em que, a partir do registrador de instrução (*IR*), o valor "01" chega até a unidade de controle (*CU*), onde é decodificado, cujo resultado é a geração de uma micro-instrução que fica à disposição do registrador de micro-instrução (*MIR*), *que* ainda não pode armazenar a nova micro-instrução, pois a sua entrada *trigger* está com valor "0"; No instante seguinte, o valor do *clock* muda de "1" para "0", e a entrada *trigger* do registrador

de micro-instrução (*MIR)* passa a receber o valor "1", onde o registrador de micro-instrução (*MIR)* armazena a micro-instrução e fornece um novo conjunto de sinais de controle nas suas saídas, onde o único sinal de controle com valor igual a "1", nas saídas do registrador de micro-instrução (*MIR),* é o *trigger_mdr*, sendo que a saída *out4* do registrador de dados da memória (*MDR)* está diretamente ligada à entrada *ir_1* do registrador de instrução (*IR),* por onde o código da instrução recebida chega ao registrador de instrução (*IR),* dito código cujo valor "1" representa uma instrução de soma, a qual é executada pelo processador no ciclo de execução;

2. um ciclo de execução que apresenta duas fases, sendo:

b.1) uma primeira fase em que o *clock* muda de valor automaticamente, passando de "0" para "1", e o *IR* recebe "1" na sua entrada *trigger*, o que o habilita a armazenar o conteúdo das linhas *ir_1* e *ir_0, que, juntas,* agora apresentam o valor "10", o qual é usado para indicar o código da próxima micro-instrução a ser fornecida pela unidade de controle (*CU),* dito valor "10" que chega até a unidade de controle (*CU)* onde é decodificado, gerando uma micro-instrução que fica à disposição do *MIR* que ainda não pode armazenar a nova micro-instrução pois a sua entrada *trigger* está com valor "0" e no instante seguinte o valor do *clock* muda de "1" para "0", com a entrada *trigger* do *MIR* passando a receber o valor "1", de forma a armazenar a micro-instrução e fornecer um novo conjunto de sinais de controle nas suas saídas *trigger_a* = 1, *trigger_b* = 1, *trigger_ic* = 1 e *ir_0* = 1, com os demais sinais de controle apresentando valores iguais a "0", com o IC apontando para o endereço da próxima instrução que somente será buscada na memória quando do término

completo do ciclo de execução, e a partir dos registradores *A* e *B*, ambos os operandos são passados para a *ALU*, onde são somados, ficando o resultado à disposição do *ACC*, nas saídas da *ALU, sendo que o* registrador *ACC* ainda não pode armazenar o resultado da soma pois a sua entrada *trigger* está com valor "0"; b.2) uma segunda fase onde o *clock* muda de valor automaticamente., passando de "0" para "1", com o *IR* recebendo "1" na sua entrada *trigger*, o que o habilita a armazenar o conteúdo das linhas *ir_1* e *ir_0 que* juntas, agora apresentam o valor "11", o qual é usado para indicar o código da próxima micro-instrução a ser fornecida pela unidade de controle *(CU) onde é decodificado e gerada* uma micro-instrução para ficar à disposição do *MIR,* onde no instante seguinte o valor do *clock* muda de "1" para "0" e a entrada *trigger* do *MIR* passa a receber o valor "1", dito registrador que consegue armazenar a micro-instrução e fornecer um novo conjunto de sinais de controle nas suas saídas, de forma que o único sinal de controle com valor igual a "1" nas saídas do *MIR* é o *trigger_acc, permitindo* ao *ACC* armazenar o resultado da soma disponibilizado pelas saídas da *ALU.*

2. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato do sinal de controle chamado *trigger_mar*, de valor "1", permitir que o registrador de endereço da memória *(MAR)* armazene o valor "00" proveniente do contador de instruções *(IC).*
3. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado

pelo fato do sinal de controle *trigger_ic*, cujo valor é "1", ser responsável por incrementar o valor do *IC*.

4. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato do sinal de controle *trigger_a*, cujo valor é "1", ser responsável por permitir que o registrador *A* armazene o primeiro operando da instrução.

5. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato do sinal de controle *trigger_b*, cujo valor é "1", ser responsável por permitir que o registrador *B* armazene o segundo operando da instrução.

6. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da segunda reconfiguração implementar uma *CU* microprogramada, descrita em VHDL, onde a implementa também os circuitos que permitem a realização das operações lógicas AND, OR, NOT e XOR, selecionadas com o auxílio de um circuito multiplexador, com decodificador do display mostrando um intervalo de 0 a 9 números decimais.

7. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da terceira reconfiguração possibilitar a reconfiguração do barramento de dados transformado em um barramento bidirecional com o fluxo de dados controlado através de um *tri-*

state e a introdução do estado de alta impedância (Z), com o código de instrução representado por dois bits também presentes no barramento de controle (como ir1 e ir0) com um tri-state adicional, um *MIR* sendo um registrador de dezesseis bits e *CU* com código VHDL documentado.

8. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da quarta reconfiguração apresentar arquitetura com quatro bits, com barramentos de dados e de endereço também de quatro bits de largura, memória *ROM* armazenando dezesseis palavras de memória com cinco bits cada uma.

9. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da quinta reconfiguração permitir realizar as operações lógicas AND, OR, NOT e XOR, além da soma, com um circuito multiplexador presente no interior da *ALU* que possibilita a seleção de uma dentre essas cinco operações, com a *CU* implementando o microcódigo necessário para controlar a execução de cada uma das operações oferecidas pela *ALU*, além do microcódigo usado para carregar os registradores *A* e *B* com dados provenientes da memória, onde o *MIR* passa a armazenar novas microinstruções e a memória *ROM* passa a armazenar dezesseis palavras de memória de quatro bits cada uma, com oito elementos *tri-state* para controlar os bits do código de microinstrução, sendo quatro implementados nas saídas do *MDR* e outros quatro dentro do *MIR*.

10. MÉTODO PARA ARQUITETURA DE COMPUTADOR

RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 9, caracterizado pelo fato da ALU comportar futuras implementações de circuitos aritméticos.

11. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da sexta reconfiguração apresentar *CU* trabalhando com micro-instruções de trinta e dois bits, onde a micromemória incorpora uma nova instrução para carregar o *ACC* diretamente com dados provenientes da memória, e um barramento interno bidirecional que interliga os registradores *MDR*, *A*, *B* e *ACC*, além da *ALU* através de elementos *tri-state, com o* fluxo de dados do barramento interno sendo controlado pela *CU* através do envio de sinais de controle para esses elementos *tri-state e MDR* apresentando saídas exclusivas para o código de instrução incorporando elementos *tri-state, e ALU* realimentada pelo *ACC*.

12. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da sétima reconfiguração apresentar dois *displays* de sete segmentos, com implementação de uma memória de armazenamento temporário do tipo *RAM* e uma memória de armazenamento permanente do tipo *Flash, ditas memória tipo RAM e Flash* trabalhando com quatro bits de entrada e quatro de saída, podem armazenar dezesseis palavras de memória cada uma e estão ligadas ao barramento de dados através de elementos *tri-state* controlados pela *CU*.

13. MÉTODO PARA ARQUITETURA DE COMPUTADOR

RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da oitava reconfiguração apresentar largura dos barramentos de dados e de endereço de oito bits.

14. MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES, de acordo com a reivindicação 1, caracterizado pelo fato da nona reconfiguração trabalhar com oito bits e até sessenta instruções básicas, implementando três contadores distintos (*IC*, *IX* e *SP)* que enviam endereços ao *MAR* através de um barramento multiplexado, com a memória *ROM* guardando sessenta e quatro palavras de memória de oito bits cada uma e a *CU* incorporando o *IR* e o *MIR*, além da micro-memória microprogramada que gera microinstruções, dita *CU* que recebe e trata um sinal de interrupção proveniente de um timer e a implementação de um registrador *PSW que* armazena o estado corrente de um programa em execução.

RESUMO

MÉTODO PARA ARQUITETURA DE COMPUTADOR RECONFIGURÁVEL E SUJEITA A CONSTANTES OTIMIZAÇÕES. Patente de invenção de um método para arquitetura de computador reconfigurável e sujeita a constantes otimizações que compreende uma arquitetura de computador implementada em FPGA (*Field Programmable Gate Array*).

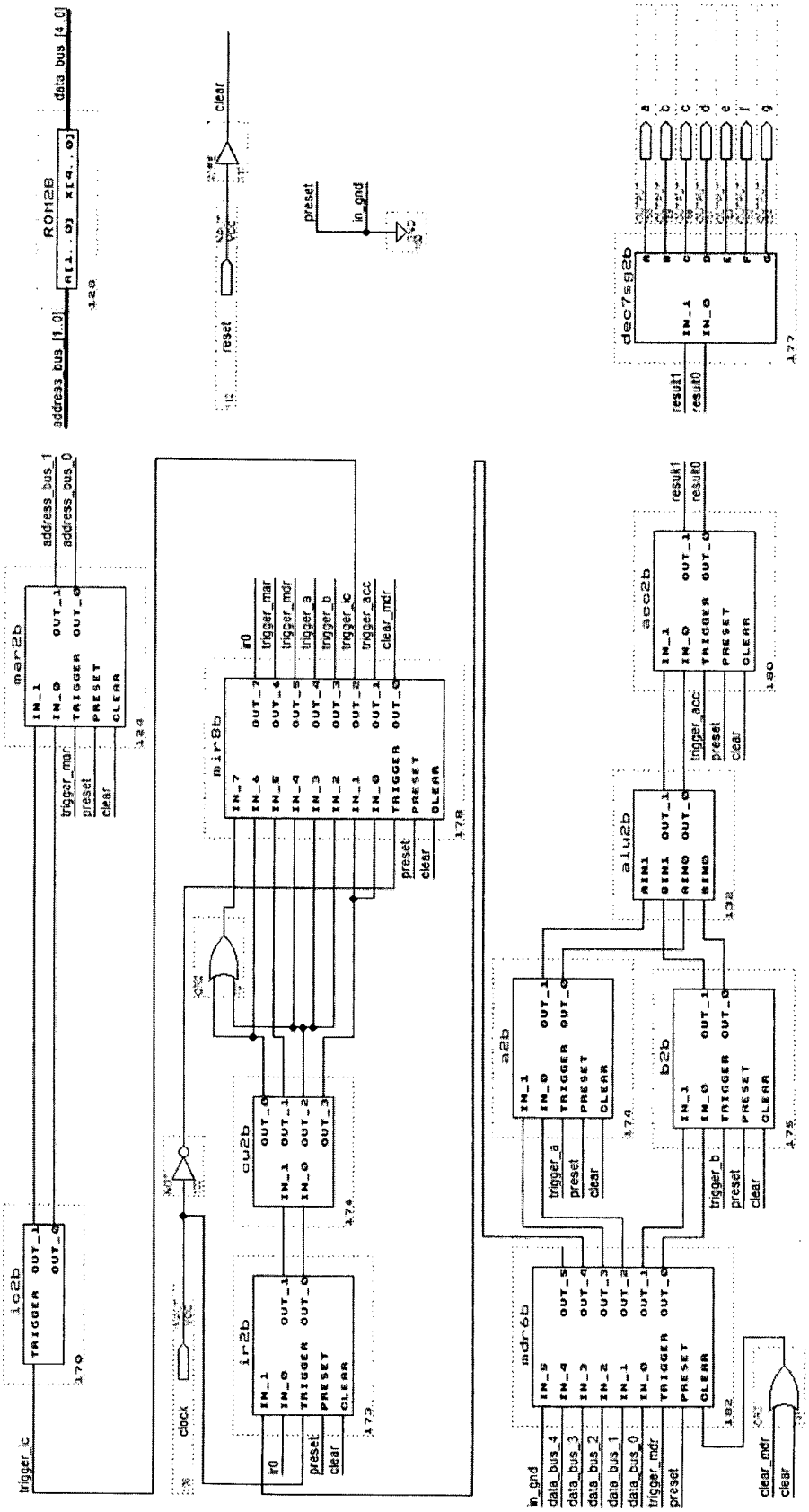

ROM2B
address_bus [1..0]
data_bus [4..0]
clear
reset
preset
in_gnd
dec7sg2b
result1
result0
mar2b
mir8b
ir0
acc2b
alu2b
a2b
b2b
ic2b
cu2b
ir2b
mdr6b
trigger_ic
clock
trigger_mar
trigger_mdr
trigger_a
trigger_b
trigger_ic
trigger_acc
clear_mdr
data_bus_4
data_bus_3
data_bus_2
data_bus_1
data_bus_0
trigger_mdr
preset
clear

Figura 1

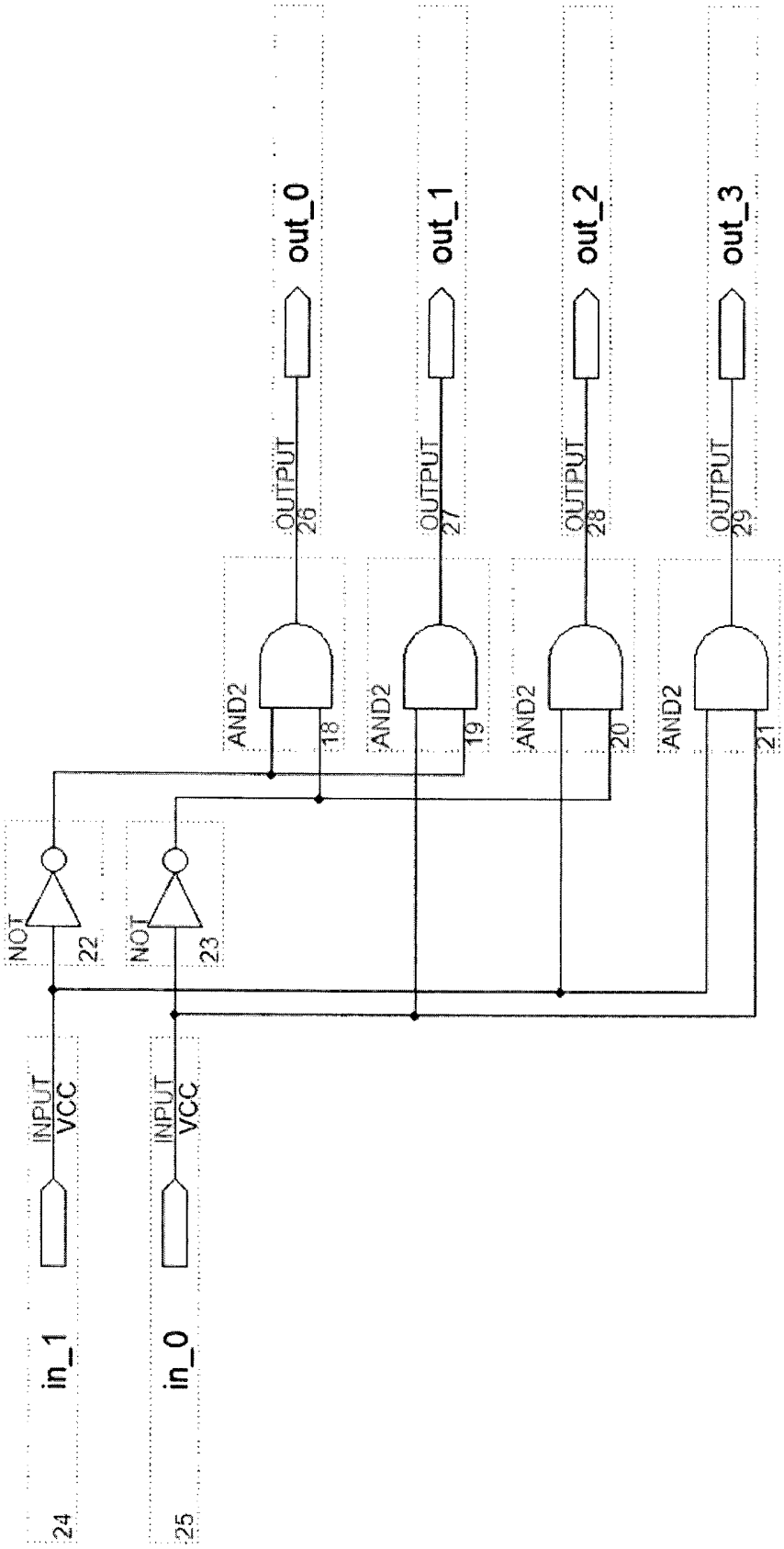

in_1
in_0
INPUT
VCC
NOT
AND2
OUTPUT
out_0
out_1
out_2
out_3
18
19
20
21
22
23
24
25
26
27
28
29

Figura 2

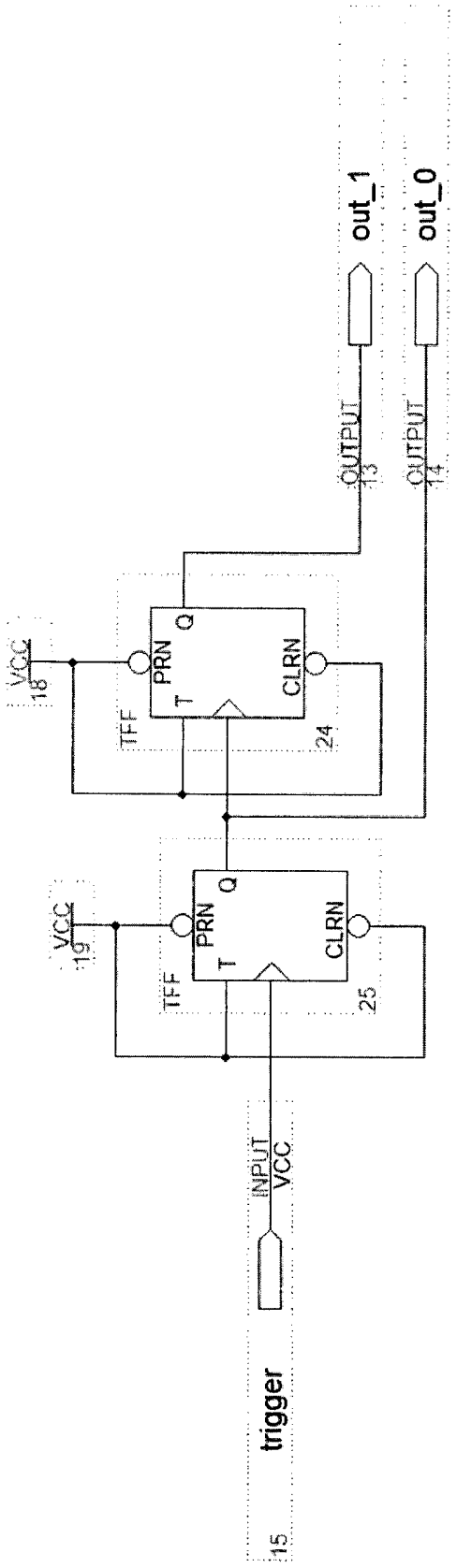

out_1
out_0
OUTPUT
13
OUTPUT
14
VCC
18
PRN
Q
T
CLRN
TFF
24
VCC
19
PRN
Q
T
CLRN
TFF
25
INPUT
VCC
trigger
15

Figura 3

```
library ieee;
use ieee.std_logic_1164.all;

entity rom2b is
    port(   a : in  std_logic_vector( 1 downto 0 );
            x : out std_logic_vector( 4 downto 0 )
        );
end rom2b;

architecture rom2bBehave of rom2b is
begin


    x <=
                "10110" when (a = "00") else
                "10101" when (a = "01") else
                "10001" when (a = "10") else
                "10010" when (a = "11");

end rom2bBehave;
```

Figura 4

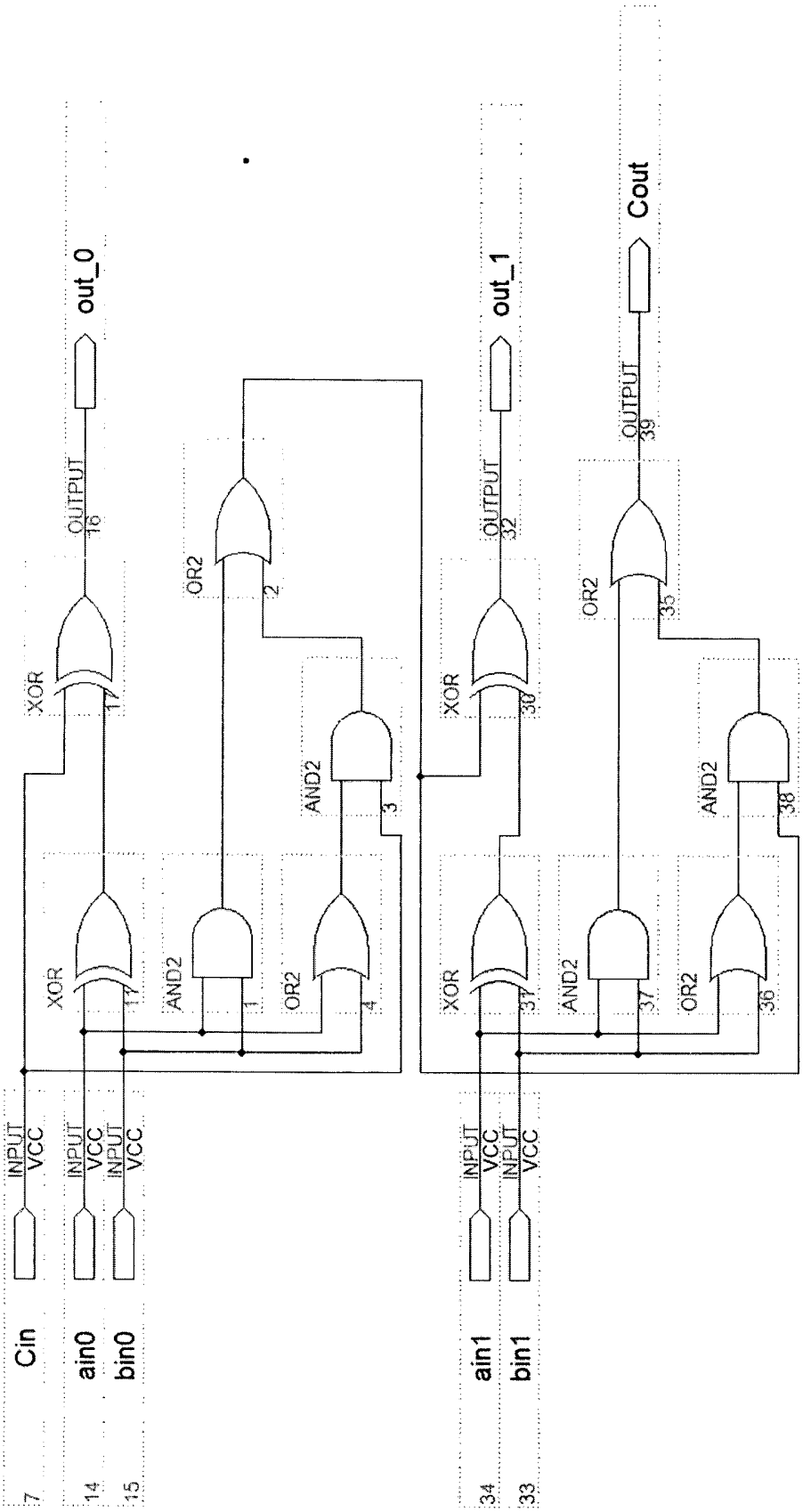

Cin
ain0
bin0
ain1
bin1
INPUT
VCC
XOR
AND2
OR2
OUTPUT
out_0
out_1
Cout

Figura 5

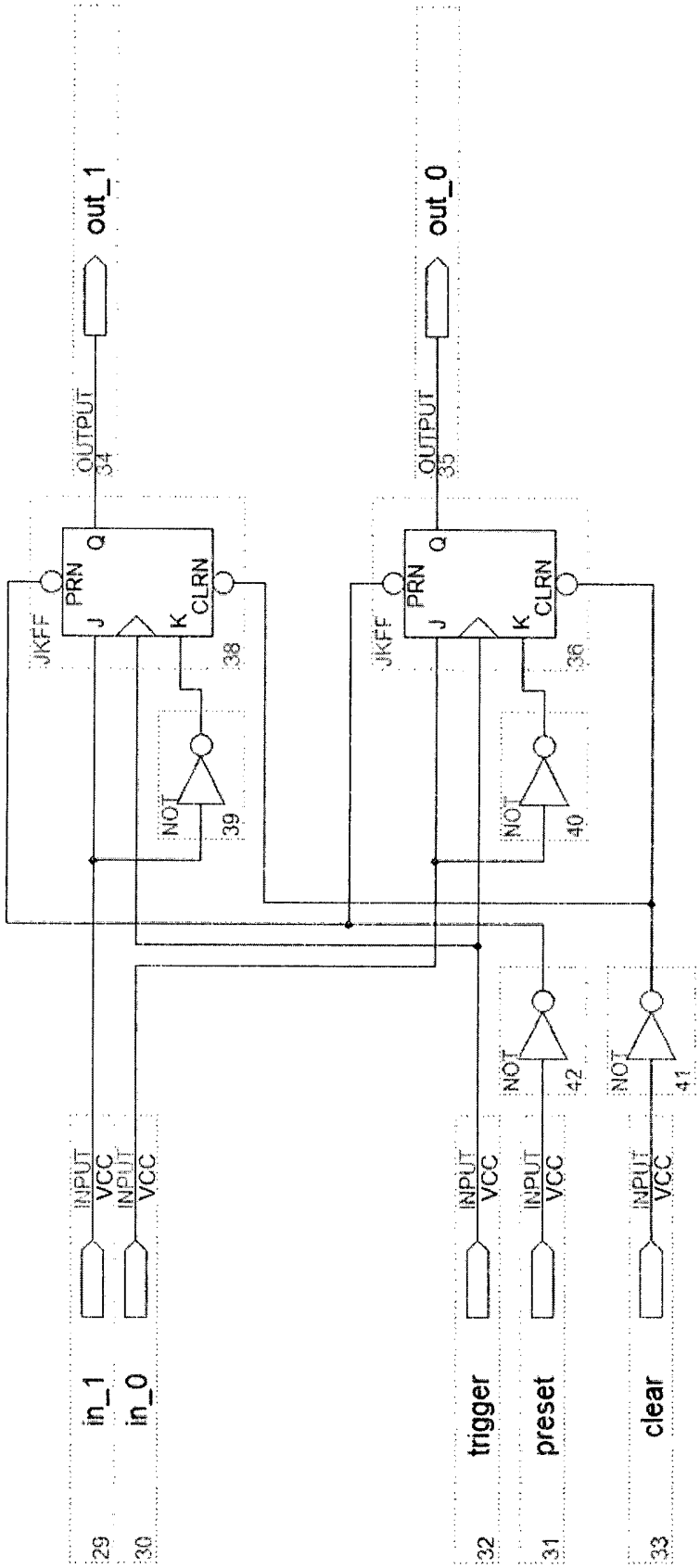

out_1
out_0
OUTPUT
34
OUTPUT
35
JKFF
PRN
Q
J
K
CLRN
38
JKFF
PRN
Q
J
K
CLRN
36
NOT
39
NOT
40
NOT
42
NOT
41
INPUT
VCC
INPUT
VCC
INPUT
VCC
INPUT
VCC
INPUT
VCC
in_1
in_0
trigger
preset
clear
29
30
32
31
33

Figura 6

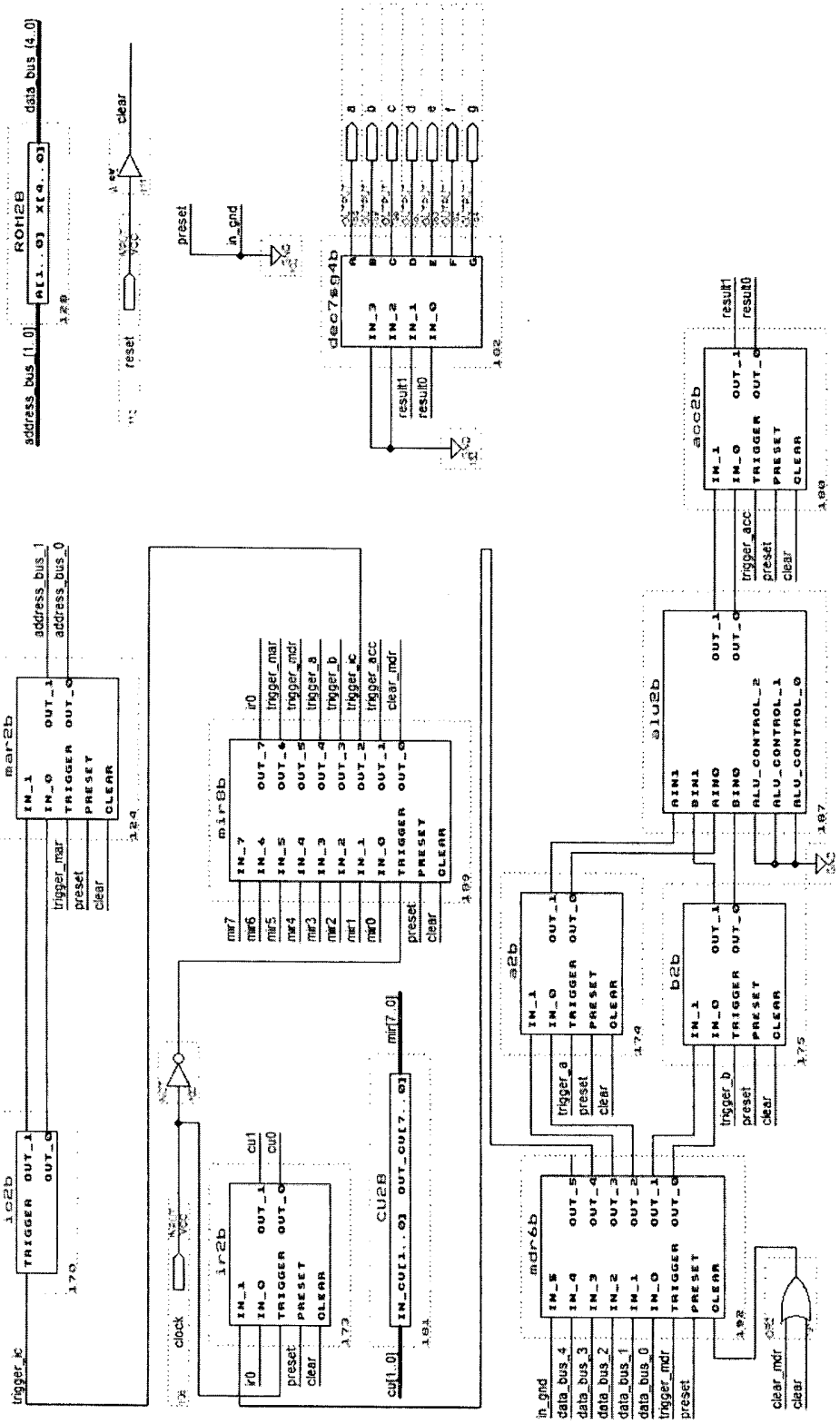

ROM128
address_bus_1[1..0]
data_bus_4[4..0]
clear
reset
preset
in_gnd
dec7seg4b
result1
result0
mar2b
ir2b
mir8b
cu2b
a2b
b2b
alu2b
acc2b
mdr6b
trigger_ic
clock
trigger_mar
trigger_mdr
trigger_a
trigger_b
trigger_acc
clear_mdr
clear_acc

Figura 7

```
library ieee;
use ieee.std_logic_1164.all;

entity cu2b is
    port(   in_cu           : in  std_logic_vector( 1 downto 0 );
            out_cu          : out std_logic_vector( 7 downto 0 )
        );
end cu2b;

architecture cu2bBehave of cu2b is
begin


    out_cu <=

-- Fetch
                "11000000" when (in_cu = "00") else
                "00100000" when (in_cu = "01") else

-- Add
                "10011100" when (in_cu = "10") else
                "00000011" when (in_cu = "11");

end cu2bBehave;
```

Figura 8

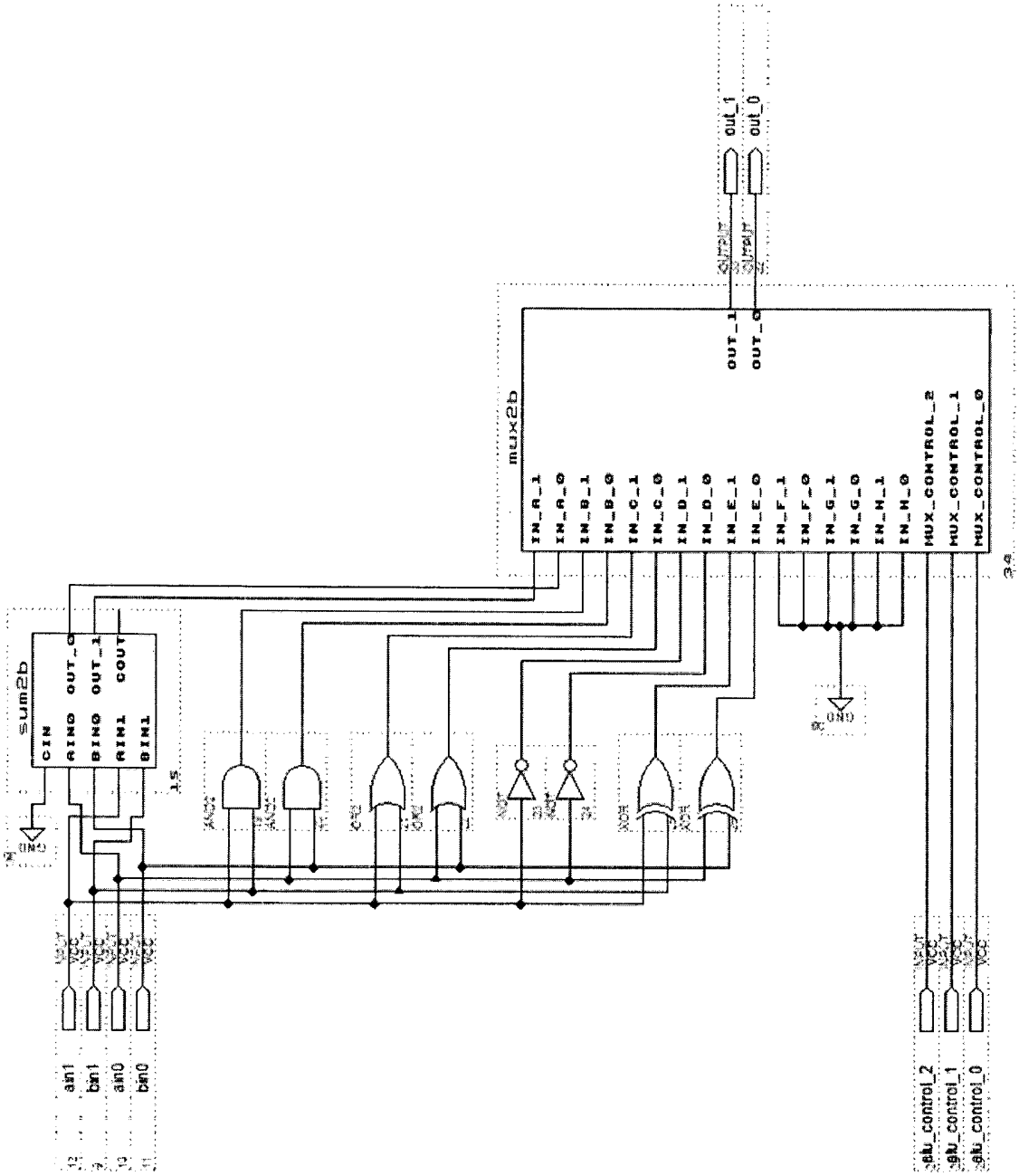

out_1
out_0
OUT_1
OUT_0
mux2b
IN_A_1
IN_A_0
IN_B_1
IN_B_0
IN_C_1
IN_C_0
IN_D_1
IN_D_0
IN_E_1
IN_E_0
IN_F_1
IN_F_0
IN_G_1
IN_G_0
IN_H_1
IN_H_0
MUX_CONTROL_2
MUX_CONTROL_1
MUX_CONTROL_0
sum2b
CIN
AIN0
BIN0
AIN1
BIN1
OUT_0
OUT_1
COUT
GND
an1
bn1
an0
bn0
mb_control_2
mb_control_1
mb_control_0

Figura 9

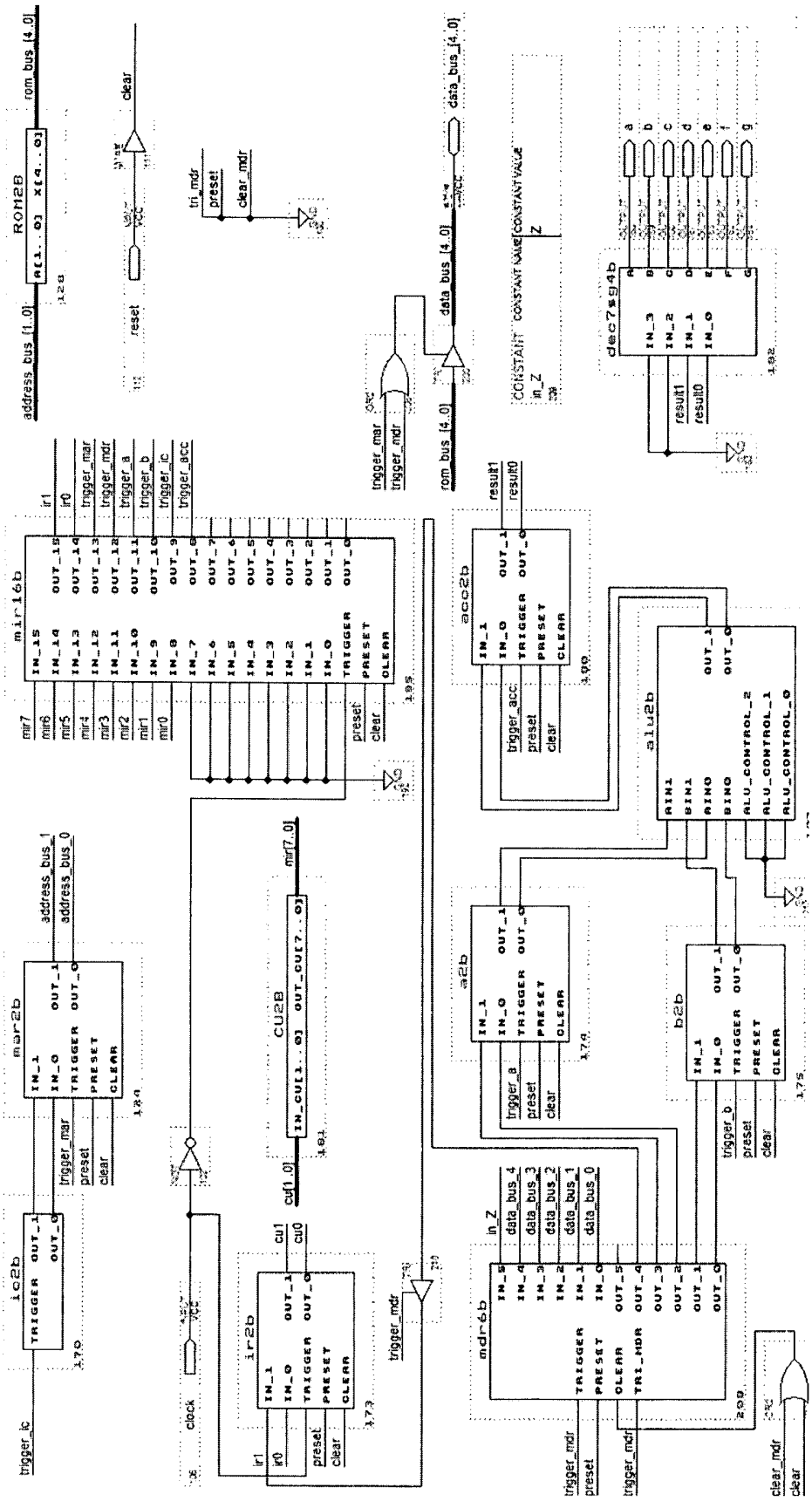

ROM2B
address_bus_[1..0]
rom_bus_[4..0]
clear
tri_mdr
preset
clear_mdr
data_bus_[4..0]
CONSTANT
CONSTANT_NAME
CONSTANT_VALUE
in_Z
dec7seg4b
result1
result0
mir16b
ir1
ir0
trigger_mar
trigger_mdr
trigger_a
trigger_b
trigger_ic
trigger_acc
mir7
mir6
mir5
mir4
mir3
mir2
mir1
mir0
acc2b
alu2b
mar2b
address_bus_1
address_bus_0
ic2b
CU2B
cu1..0
mir7..0
ir2b
cu1
cu0
a2b
b2b
mdr4b
data_bus_4
data_bus_3
data_bus_2
data_bus_1
data_bus_0
clock
trigger_ic
trigger_mdr
clear_mdr

Figura 10

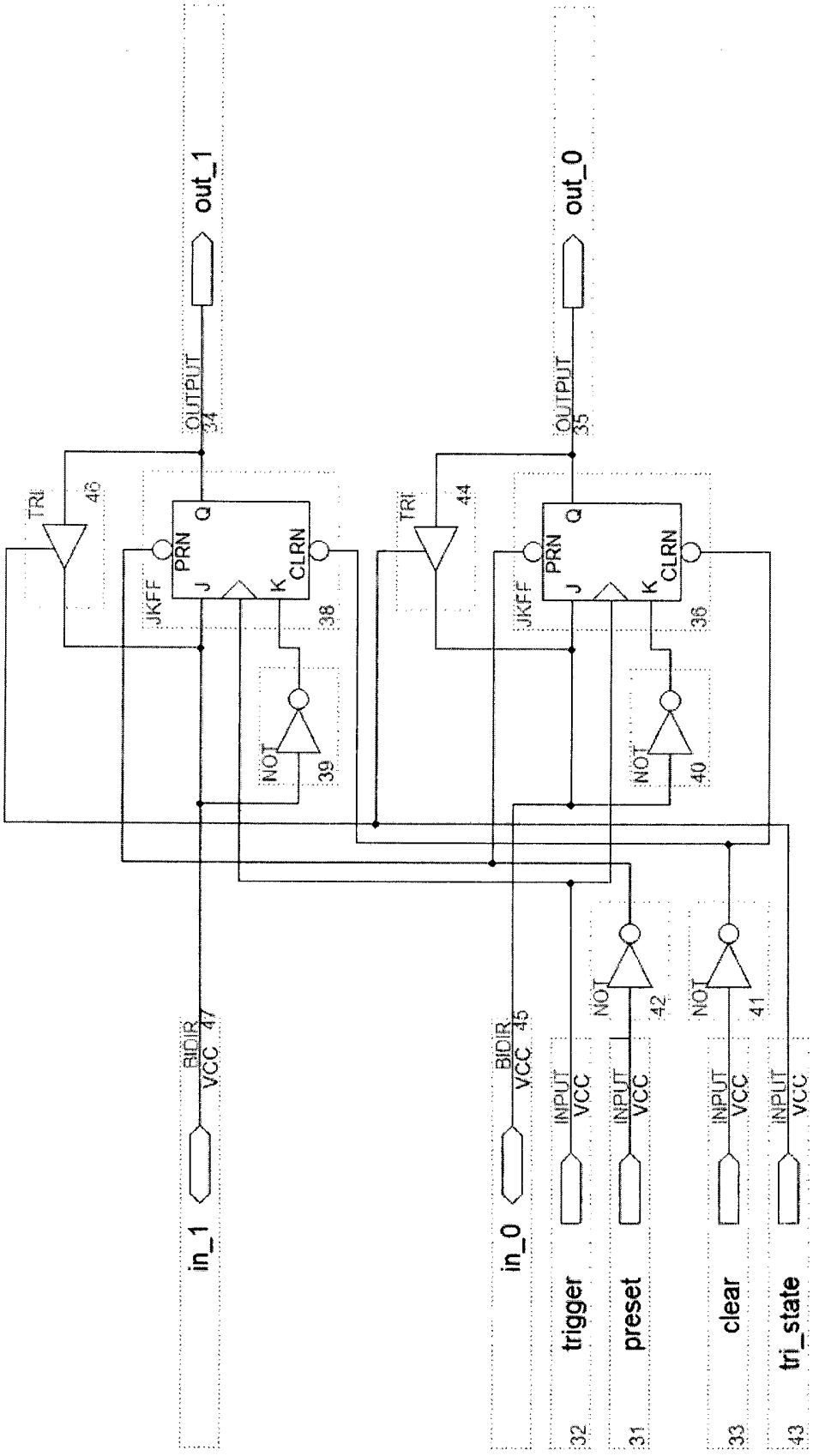

out_1
OUTPUT
34
out_0
OUTPUT
35
TRI
46
TRI
44
JKFF
PRN
Q
J
K
CLRN
38
38
NOT
39
NOT
40
NOT
42
NOT
41
BIDIR
VCC
47
BIDIR
VCC
45
INPUT
VCC
INPUT
VCC
INPUT
VCC
INPUT
VCC
in_1
in_0
trigger
preset
clear
tri_state
32
31
33
43

Figura 11

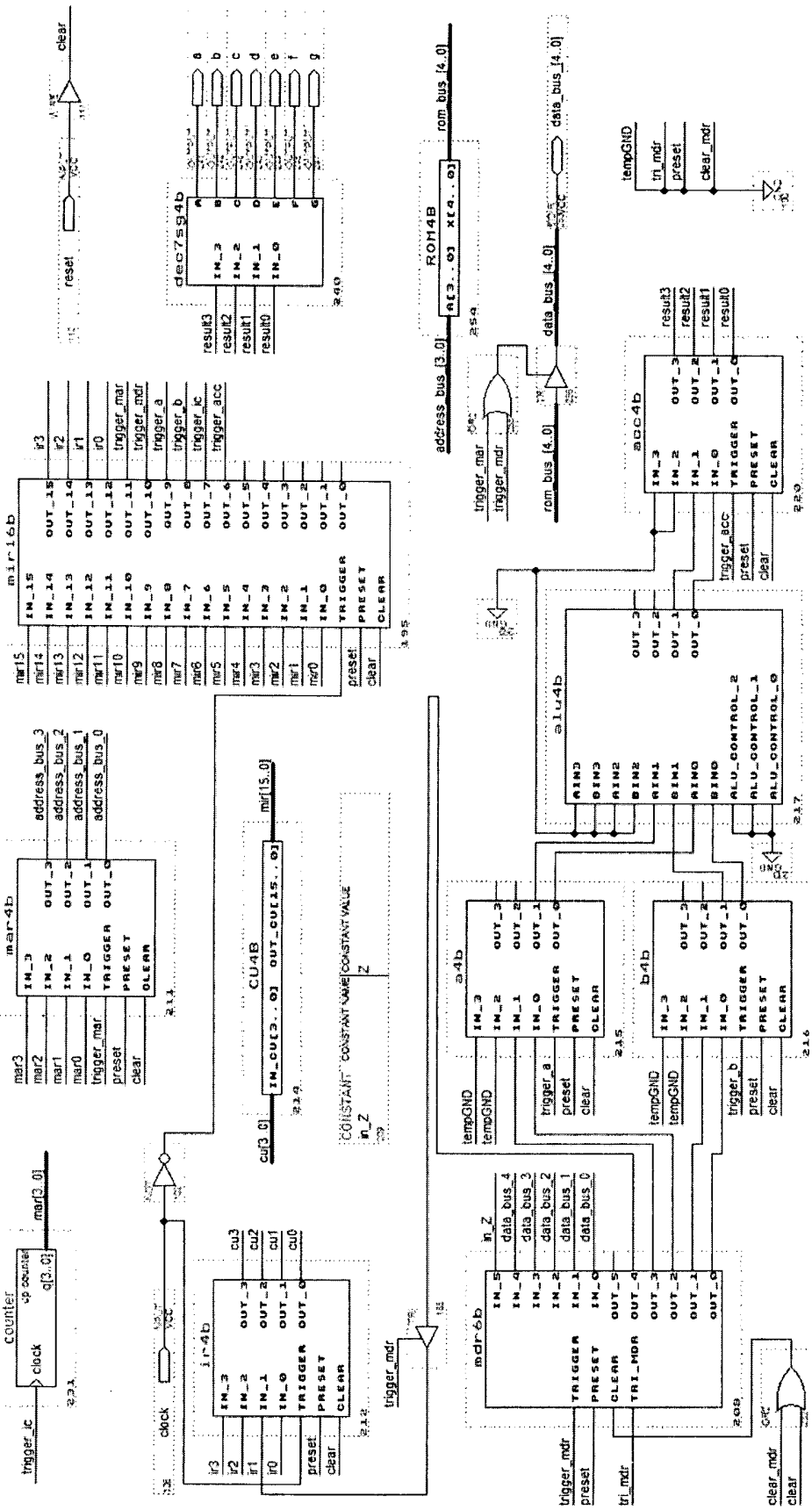

counter
up counter
clock
q[3..0]
mar3_0]
trigger_ic
ir4b
IN_3
IN_2
IN_1
IN_0
TRIGGER
PRESET
CLEAR
OUT_3
OUT_2
OUT_1
OUT_0
cu3
cu2
cu1
cu0
ir3
ir2
ir1
ir0
preset
clear
trigger_mdr
mar4b
address_bus_3
address_bus_2
address_bus_1
address_bus_0
trigger_mar
CU4B
IN_CU[3..0]
OUT_CU[15..0]
cu[3..0]
mar[15..0]
CONSTANT
CONSTANT_NAME[CONSTANT_VALUE]
in_Z
mir16b
mir15
mir14
mir13
mir12
mir11
mir10
mir9
mir8
mir7
mir6
mir5
mir4
mir3
mir2
mir1
mir0
trigger_a
trigger_b
trigger_ic
trigger_acc
trigger_mar
trigger_mdr
dec7seg4b
result3
result2
result1
result0
a
b
c
d
e
f
g
reset
vcc
clear
ROM4B
address_bus_[3..0]
rom_bus_[4..0]
data_bus_[4..0]
trigger_mar
trigger_mdr
tempGND
tri_mdr
preset
clear_mdr
acc4b
trigger_acc
alu4b
AIN3
AIN2
AIN1
AIN0
BIN3
BIN2
BIN1
BIN0
ALU_CONTROL_2
ALU_CONTROL_1
ALU_CONTROL_0
a4b
b4b
mdr4b
TRI_MDR
data_bus_4
data_bus_3
data_bus_2
data_bus_1
data_bus_0

Figura 12

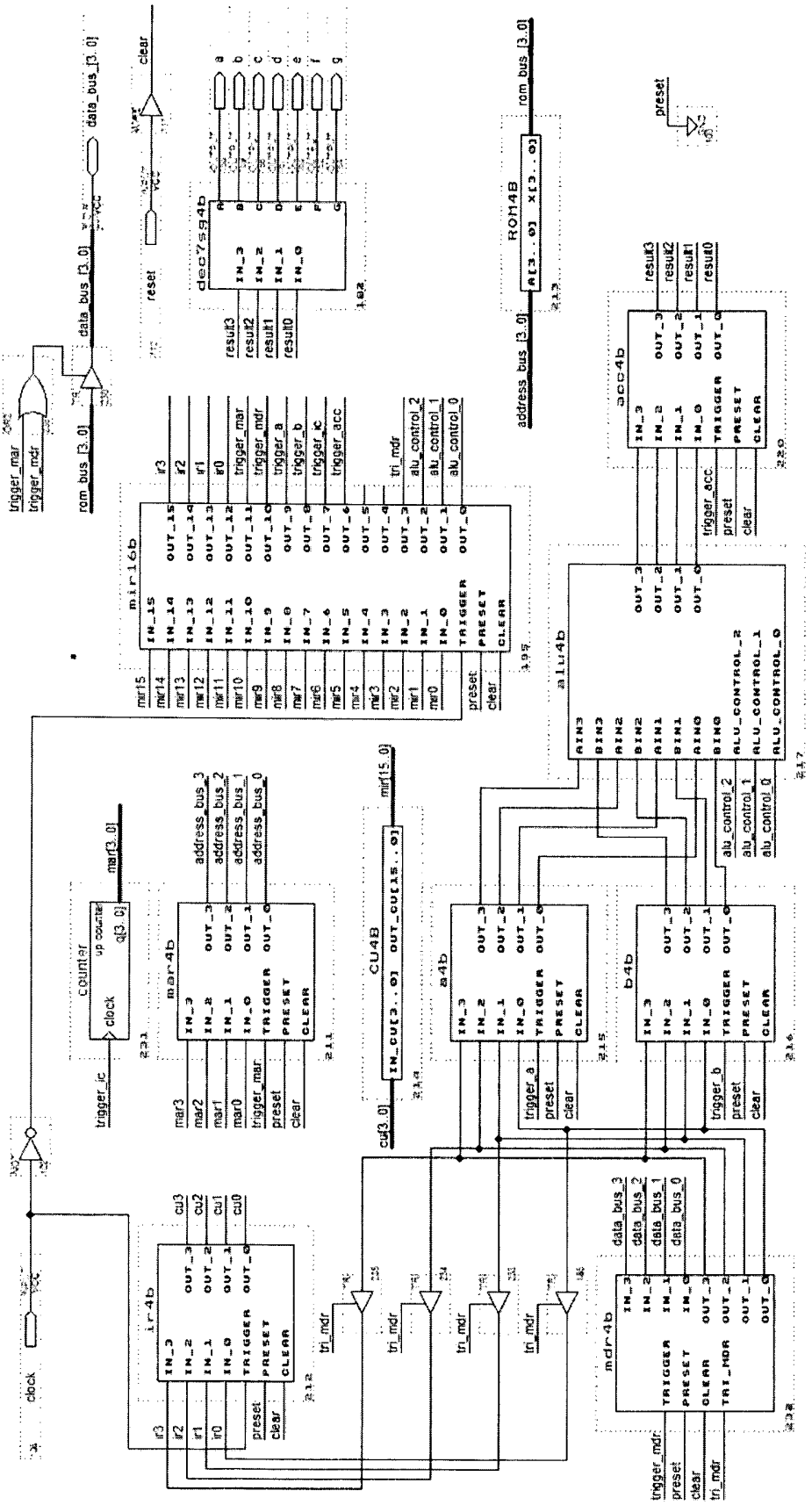

clock
trigger_mar
trigger_mdr
rom_bus_[3..0]
data_bus_[3..0]
clear
reset
preset
address_bus_[3..0]
ROM4B
A[3..0]
X[3..0]
decYsg4b
result3
result2
result1
result0
mir16b
ir3
ir2
ir1
ir0
trigger_mar
trigger_mdr
trigger_a
trigger_b
trigger_ic
trigger_acc
tri_mdr
alu_control_2
alu_control_1
alu_control_0
counter
up counter
clock
q[3..0]
trigger_ic
mar3_0]
mar-4b
mar3
mar2
mar1
mar0
address_bus_3
address_bus_2
address_bus_1
address_bus_0
trigger_mar
preset
clear
CU4B
IN_CU[3..0]
OUT_CU[15..0]
mir[15..0]
cu[3..0]
a4b
b4b
trigger_a
trigger_b
acc4b
result3
result2
result1
result0
trigger_acc
alu4b
AIN3
BIN3
AIN2
BIN2
AIN1
BIN1
AIN0
BIN0
ALU_CONTROL_2
ALU_CONTROL_1
ALU_CONTROL_0
ir4b
cu3
cu2
cu1
cu0
mdr4b
data_bus_3
data_bus_2
data_bus_1
data_bus_0
trigger_mdr
TRI_MDR

Figura 13

```
library ieee;
use ieee.std_logic_1164.all;

entity cu4b is
    port(   in_cu           : in  std_logic_vector( 3 downto 0 );
            out_cu          : out std_logic_vector( 15 downto 0 )
        );
end cu4b;

architecture cu4bBehave of cu4b is
begin

    out_cu <=
-- Fetch
                "0001100000000000" when (in_cu = "0000") else   -- trigger_mar
                "0000010010001000" when (in_cu = "0001") else   -- trigger_mdr; trigger_ic; tri_mdr
-- Load a4b
                "0011100000000000" when (in_cu = "0010") else   -- trigger_mar
                "0100010010000000" when (in_cu = "0011") else   -- trigger_mdr; trigger_ic
                "0000001000000000" when (in_cu = "0100") else   -- trigger_a
-- Load b4b
                "0110100000000000" when (in_cu = "0101") else   -- trigger_mar
                "0111010010000000" when (in_cu = "0110") else   -- trigger_mdr; trigger_ic
                "0000000100000000" when (in_cu = "0111") else   -- trigger_b
-- Add
                "1001000000000000" when (in_cu = "1000") else   -- NOP
                "0000000001000000" when (in_cu = "1001") else   -- trigger_acc; alu_control = "00"
-- And
                "1011000000000001" when (in_cu = "1010") else   -- NOP
                "0000000001000001" when (in_cu = "1011") else   -- trigger_acc; alu_control = "01"
-- Or
                "1101000000000010" when (in_cu = "1100") else   -- NOP
                "0000000001000010" when (in_cu = "1101") else   -- trigger_acc; alu_control = "10"
-- Not a4b
                "1111000000000011" when (in_cu = "1110") else   -- NOP
                "0000000001000011" when (in_cu = "1111");       -- trigger_acc; alu_control = "11"
end cu4bBehave;
```

Figura 14

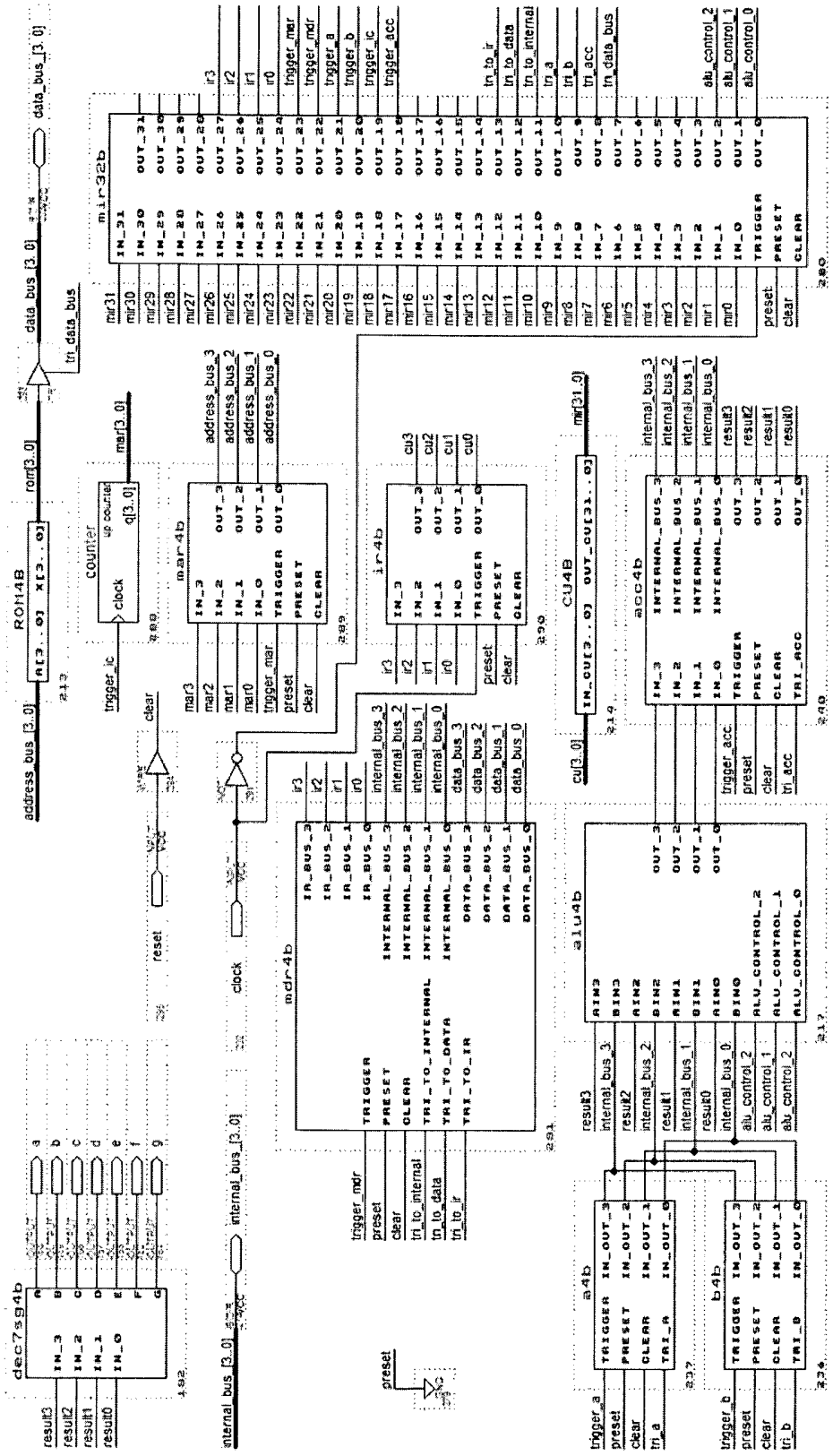

ROM4B
counter
mar4b
mir32b
ir4b
CU4B
acc4b
mdr4b
alu4b
a4b
b4b
dec7sg4b
address_bus_[3..0]
data_bus_[3..0]
tri_data_bus
internal_bus_[3..0]
clock
reset
preset

Figura 15

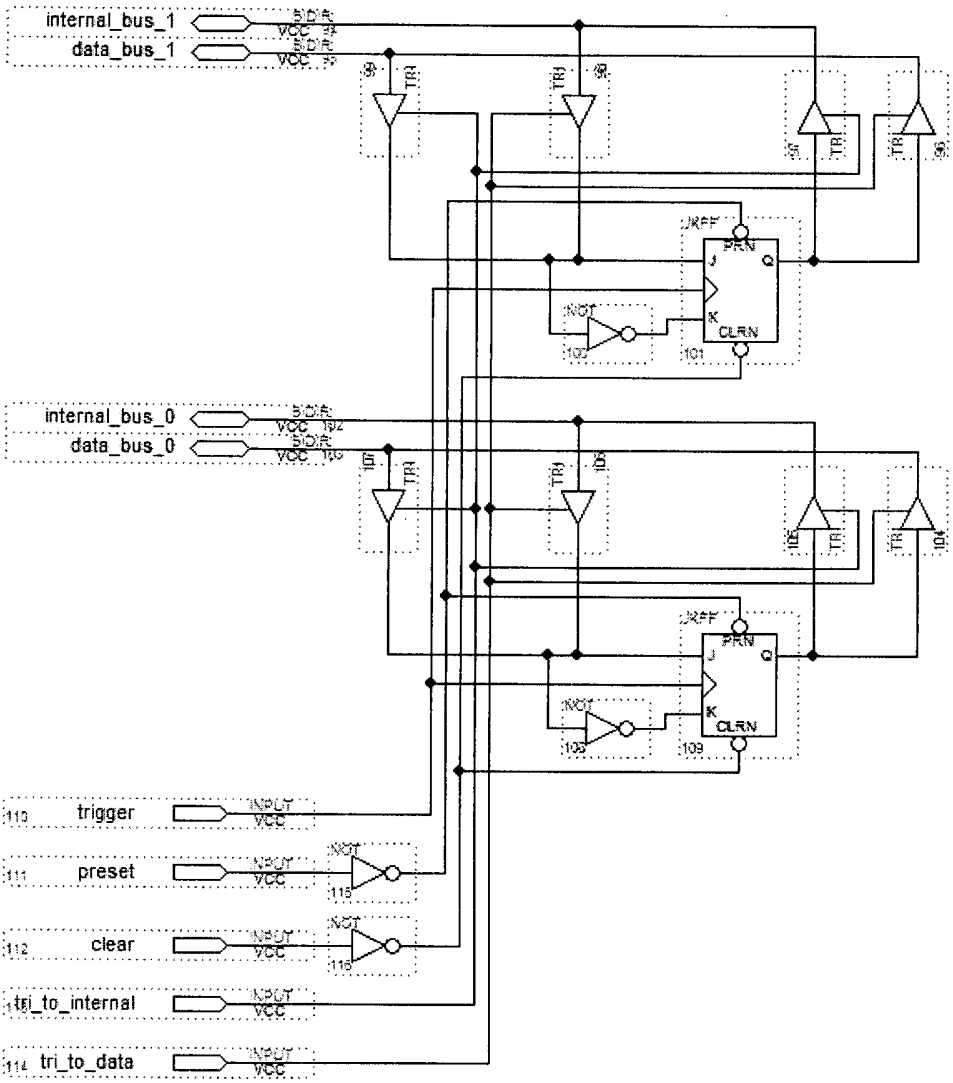

internal_bus_1
data_bus_1
internal_bus_0
data_bus_0
trigger
preset
clear
tri_to_internal
tri_to_data
INPUT
VCC
TRI
JKFF
PRN
CLRN
NOT

Figura 16

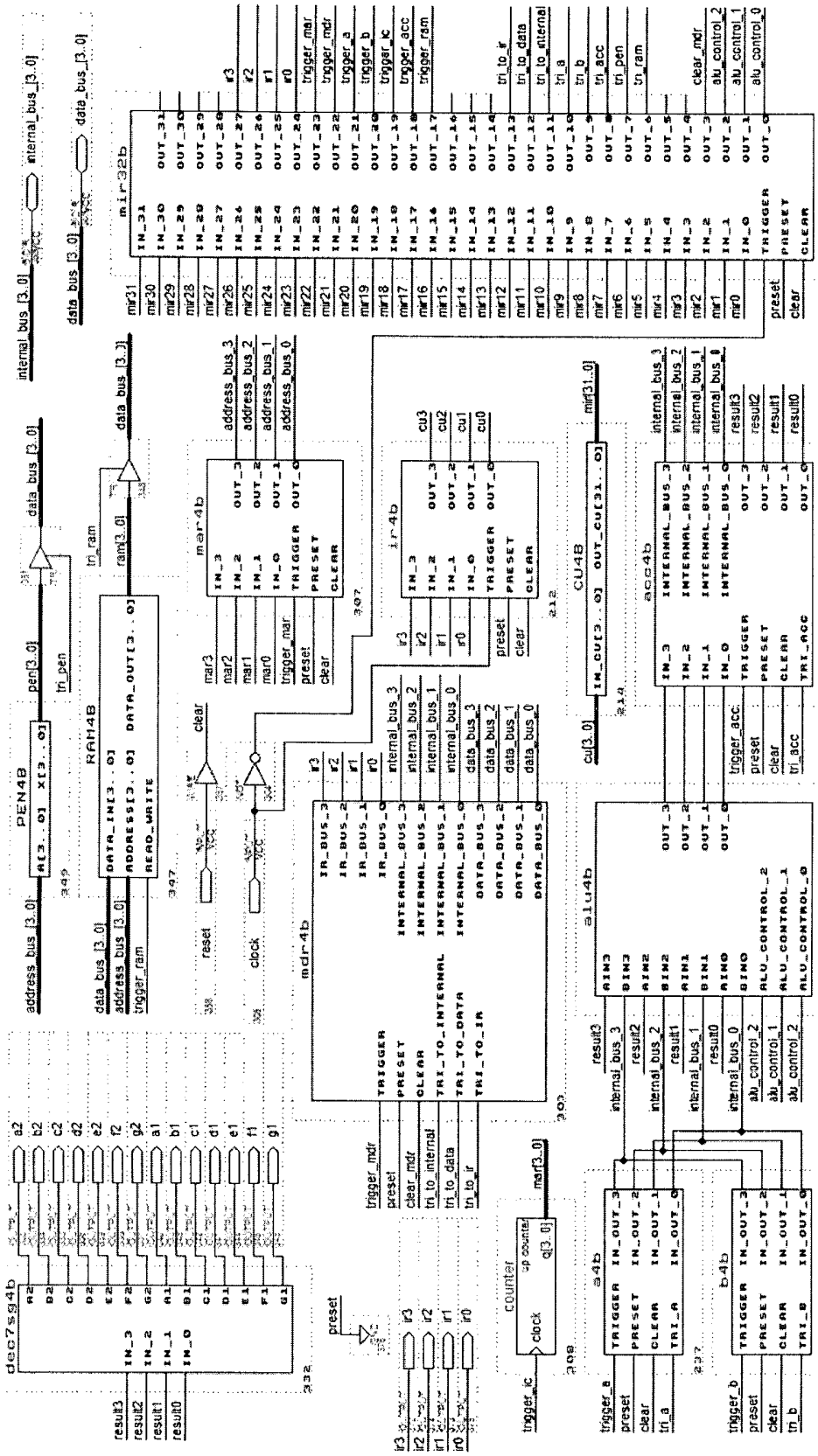


Figura 17

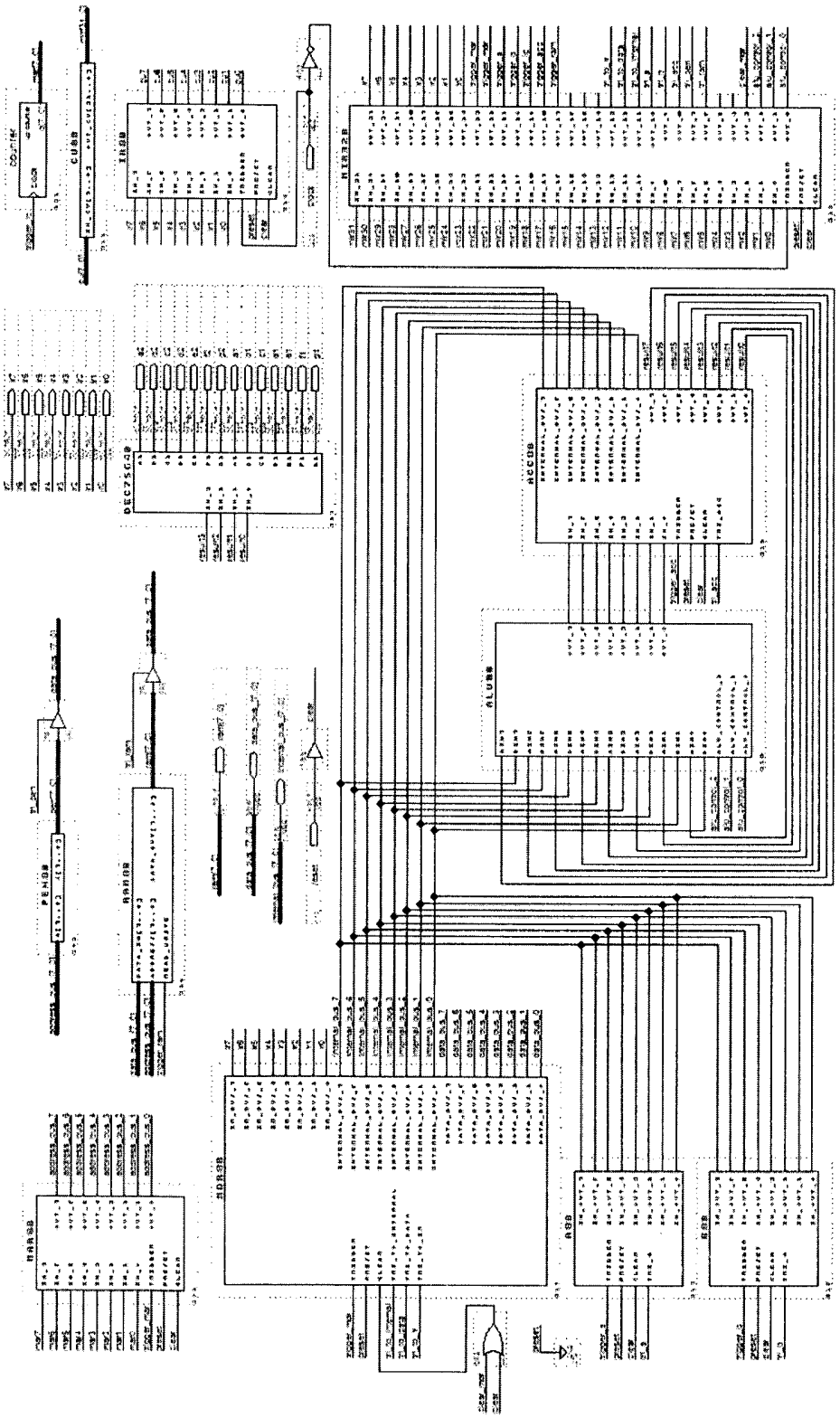


Figura 18

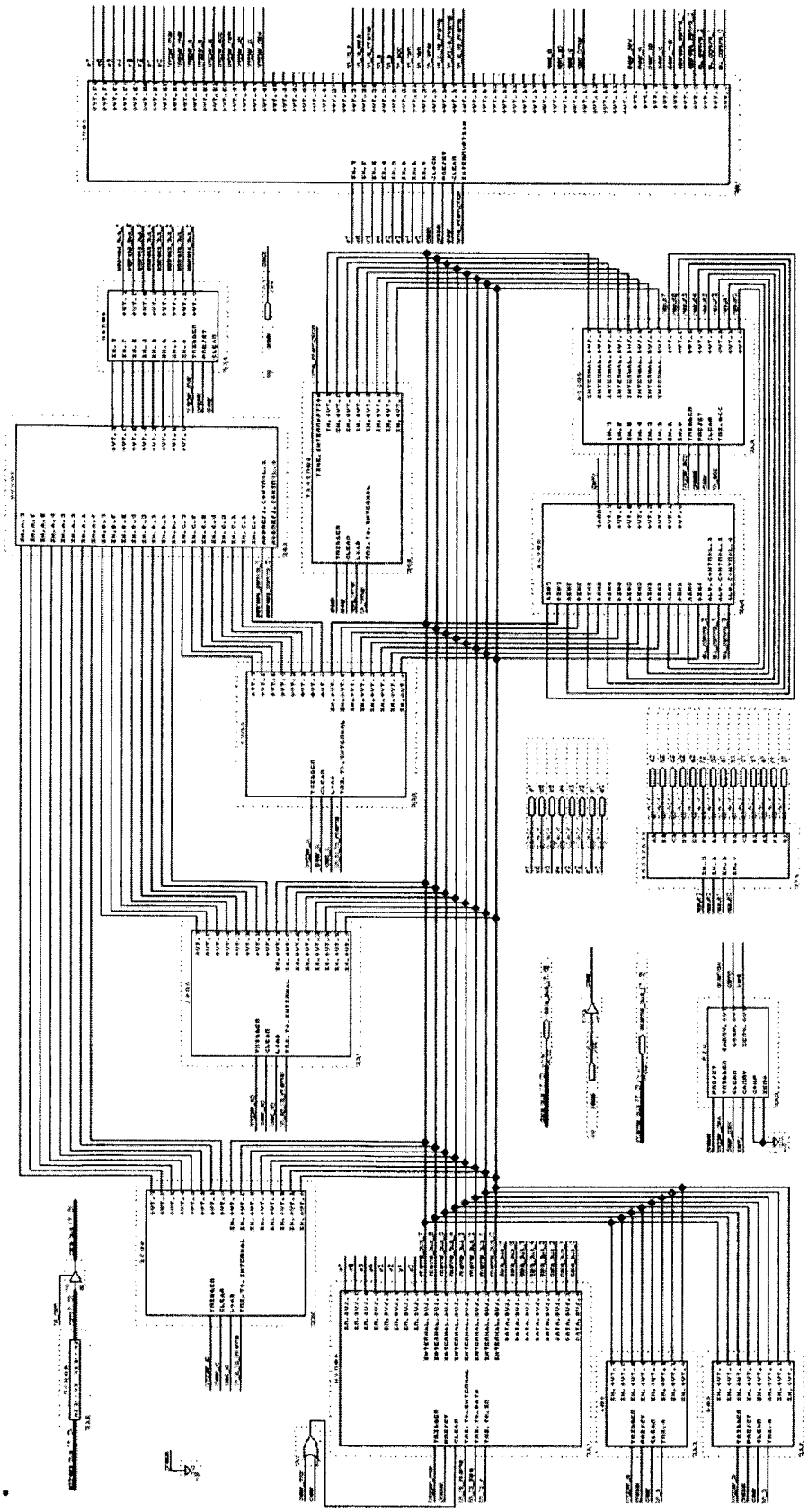


Figura 19