Processadores de Instrução Única

Arquiteturas URISC/OISC

Arquitetura de Computadores

O Conceito OISC

- **O que é:** Classe de processadores que opera somente com uma única instrução universal.

- **Surgimento:** Primeira formalização do conceito de um processador de instrução única no artigo "URISC: The Ultimate Reduced Instruction Set Computer" (1988), por Farhad Mavaddat e Behrooz Parhami.

- **Evolução:** Após a formalização, o conceito ganhou diferentes variantes e paradigmas que melhoraram e/ou modificaram a implementação original, mais conhecidos atualmente como OISC (One Instruction Set Computer).

Motivações

Turing Completude

Demonstração de que um processador de uma única instrução pode ser Turing completo, possuindo o mesmo poder computacional de uma Máquina de Turing.

Simplicidade

Diferente das arquiteturas comerciais, simplifica a estrutura do hardware e passa a complexidade do funcionamento para o software.

Educação

Apesar da simplicidade da ISA, a implementação exige a compreensão de todos os componentes vitais de um computador real: PC, ULA, barramentos, etc.

Mas como?

- **Instrução atômica:** Uma única instrução realiza transferência de dados, cálculo e controle, tudo em um único ciclo.

- **Sem Opcode:** Como há apenas uma instrução, a necessidade de decodificação de instruções é removida.

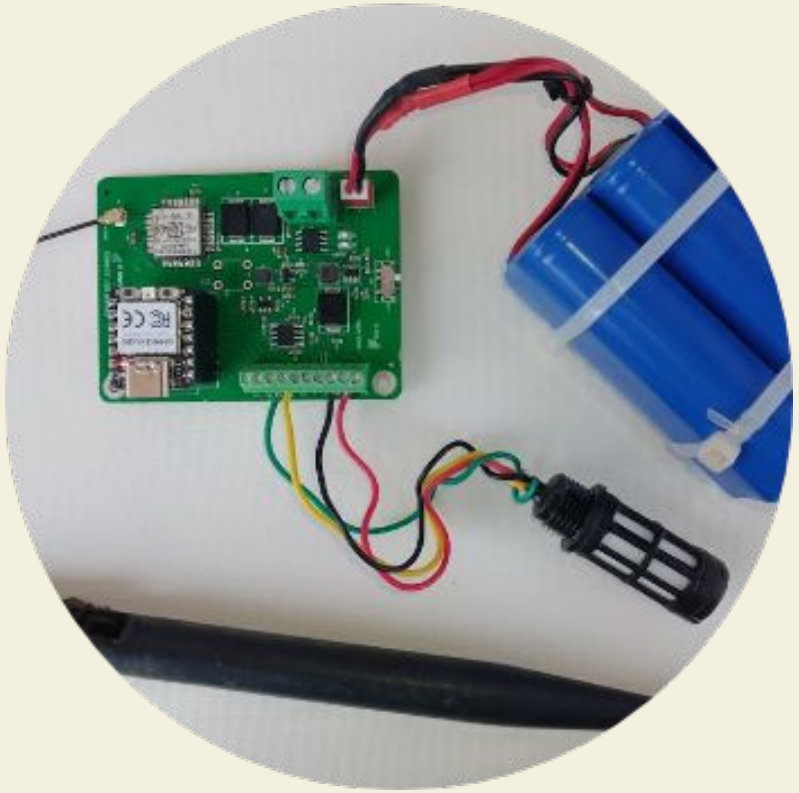
- **Universalidade:** A escolha de uma operação específica permite a síntese de todas as outras operações lógicas e aritméticas necessárias para a universalidade computacional

Tipos de processadores de uma instrução

- **Máquinas de manipulação de Bit:** Modelo mais simples de OISC, que utiliza manipulação de bits para realizar operções lógicas e aritméticas e controle de fluxo. Ex: FlipJump
- **Arquitetura Habilitada por Transporte (TTA):** Utiliza apenas o MOV entre registradores para computar informações, realizando operações diferentes a cada registrador acessado.
- **Máquinas baseadas em Aritmética:** Utilizam operações aritméticas e um pulo condcional, operando com números inteiros, que podem representar endereços na memória. Ex.: Subleq

Baseado nessas características, você consegue imaginar cenários reais de utilização dessa arquitetura?

Aplicações Práticas: IoT



- Baixo consumo energético
- Processamento não recorrente
- Hardware limitado ao extremo
- Aplicações de baixo custo
- Conectividade mínima

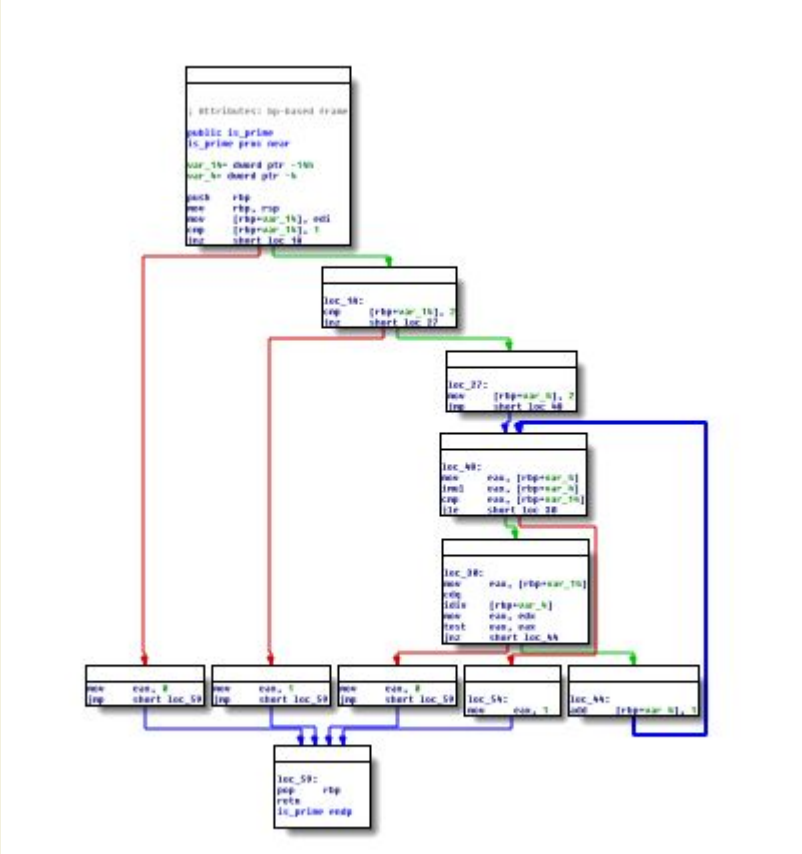
Aplicações Práticas: Criptografia

- **Ofuscação de código**
  - Ausência de desvios no fluxo de execução
  - Dificuldade em encontrar padrões
- **Criptografia Homomórfica**
  - Computar dados sem descriptografá-los
- **Criação de ambientes de execução rigidamente seguros**
- **Exemplos**
  - MOVfuscator
  - CryptoLeq

Código compilado utilizando o MOVfuscator

Aplicações Práticas: Criptografia

Diagramas de fluxo para um programa que determina se um número é primo



Código compilado utilizando o GCC padrão

Código compilado utilizando o MOVfuscator

Aplicações Práticas: Coprocessamento

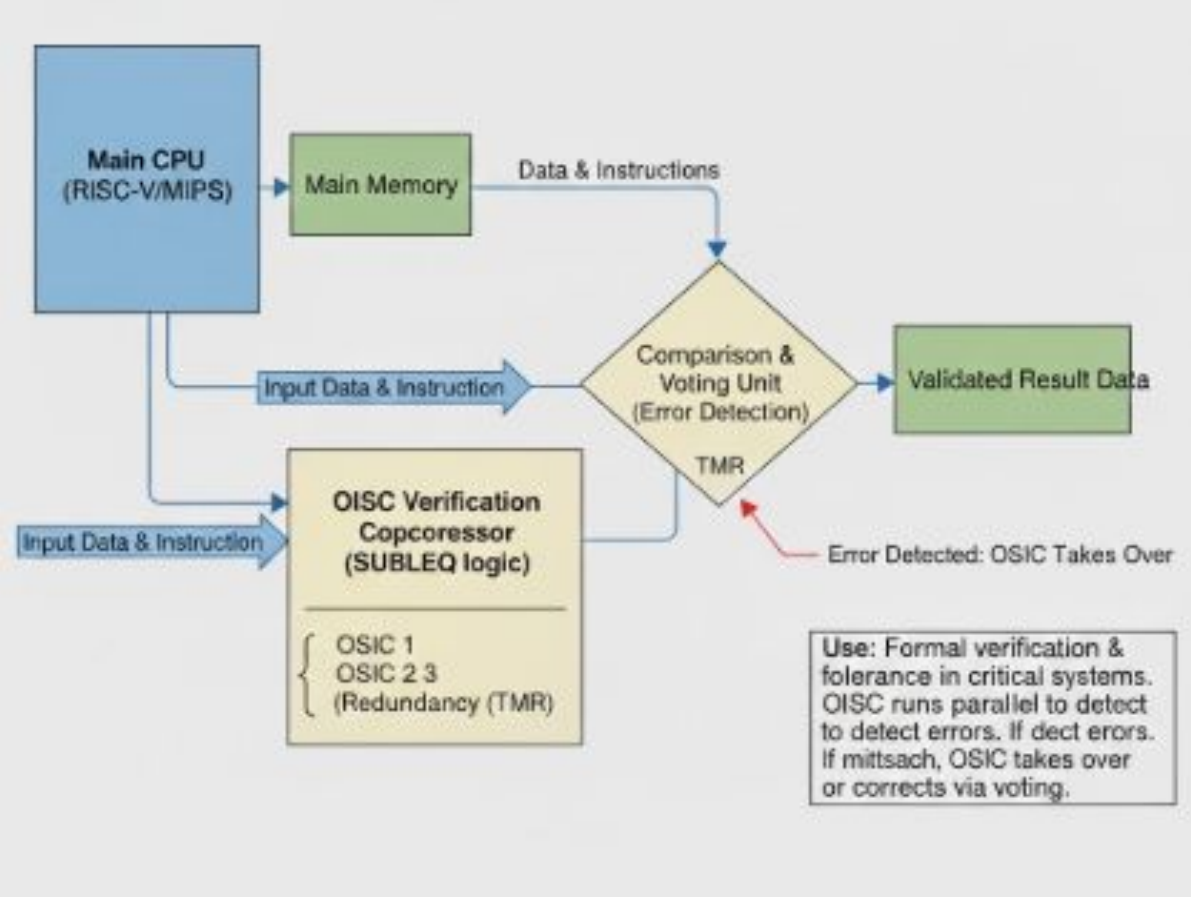


Diagrama de coprocessamento com CPU OISC

Uso como coprocessadores de verificação formal para **detectar falhas** em processadores mais complexos, como MIPS ou RISC-V, em **sistemas críticos.** Se o processador principal falha ou apresenta um erro, o coprocessador OISC assume a execução daquela instrução específica via software.

- **Processamento especializado**
- **Execução em paralelo**
- **Aplicações de baixo custo**

Aplicações Práticas: Nanotecnologia

- **Reduzir complexidade física:** Menos tipos de instrução → Menos lógica → Melhor viabilidade em escalas nanométricas.

- **Viabilidade em Estágios Experimentais:** Permite a implementação de processamento de dados em tecnologias pós-silício que ainda possuem baixa densidade de integração.

- **Aceleração de Pesquisa e Desenvolvimento:** A simplicidade do hardware permite a validação de conceitos de forma rápida.

"Em nanoescala, não é a sofisticação da instrução que importa, mas a simplicidade da física que a executa."

Lado negativo do OISC

- **Desempenho lento:** Devido à necessidade de emular instruções complexas com muitas operações simples, o desempenho é significativamente inferior aos processadores tradicionais.

- **Expansão de código:** Um programa simples pode se tornar imenso ao ser convertido para um conjunto de uma única instrução.

- **Dificuldade de depuração:** Por ser um bloco único de código, o software de um processador OISC se torna complexo e mais difícil de se aplicar correções e/ou alterações.

O SUBLEQ

A instrução mais comum para OISCs é a **SUBLEQ** (Subtract and Branch if Less or Equal to Zero), que combina aritmética e controle de fluxo em uma única instrução, da seguinte forma:

- Dados 3 operandos A, B e C, e a instrução executada no formato *A B C*:
  - Realiza a operação [B] − [A] e armazena em B.
  - Se [B] ≤ 0, desvia para C; senão, continua na próxima instrução (PC+3)
  - Caso PC = −1, o programa é encerrado

```
mem [ b ]  = mem [ b ]  - mem [ a ]    se  mem [ b ]  ≤ 0 ,   pule para  c
```

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 0 0 -10
18: -1 0 0
```

Estado inicial: **A linha 0 zera os elementos no endereço 15**

O endereço 15 servirá como variável temporária.

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 10 0 -10
18: -1 0 0
```

Primeiro passo. [17] (–10) é decrementado de [15] (0).

[17] será o oposto do valor máximo do nosso contador

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 10 0 -10
18: -1 0 0
```

Segundo passo:[16] (0) é decrementado de [15] (10).

O valor em 16 será nosso contador

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 10 1 -10
18: -1 0 0
```

terceiro passo: O [18] (-1) é decrementado de[16] (0).

O valor em 16 se torna 1

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 0 1 -10
18: -1 0 0
```

Quarto passo: Enquanto o algoritmo chegar ao endereço 12, ele voltará ao início

Isso é o equivalente a uma instrução de JUMP

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 0 1 -10
18: -1 0 0
```

Quinto passo: O ciclo se repete novamente

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 10 1 -10
18: -1 0 0
```

Sexto passo: O [15] volta, novamente, a ser 10

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 9 1 -10
18: -1 0 0
```

Sétimo passo: O [16] (1) é subtraído de [15].

[15] agora é 9

O SUBLEQ: Exemplo

[X] = valor em X

X = endereço X

Vamos analisar o programa a seguir:

```
00: 15 15 3
03: 17 15 6
06: 16 15 -1
09: 18 16 12
12: 15 15 0
15: 9 2 -10
18: -1 0 0
```

Oitavo passo: [16] é incrementado e se torna 2...

E assim por diante, até chegar a 10.

O SUBLEQ (com impressão)

Uma variante do SUBLEQ, implementada aqui, possui uma função a mais:

Dada a instrução `A B C`, Caso B receba o valor -1, o caractere correspondente ao valor de A é impresso na tela

O SUBLEQ (com impressão)

```
15 17 -1
17 -1 -1
16 1 -1
16 3 -1
15 15 0
0 -1 104
116 116 112
115 58 47
47 121 111
117 116 117
46 98 101
```

Parte 1

```
47 100 81
119 52 119
57 87 103
88 99 81
63 115 105
61 121 95
107 57 76
56 83 103
112 121 56
74 85 104
103 95
```

Parte 2

Exemplo de código que imprime texto utilizando o subleq (qual será o texto?)

Funcionamento

Operação Desejada	Macro em SUBLEQ	Explicação Lógica
Zerar b	subleq b, b	Subtrai o valor dele mesmo, resultando em zero.
Adição (b = b + a)	subleq a, t; subleq t, b	Subtrai 'a' de um temporário e então subtrai o negativo de 'b'.
Mover (b = a)	subleq b, b; subleq a, t; subleq t, b	Zera 'b' e adiciona 'a'.
Salto Incondicional	subleq z, z, alvo	Como 0 ≤ 0 é sempre verdade, o salto sempre ocorre.

OISC Pedagógico (Drexel University)

Visão Geral da Arquitetura:

- **Base Teórica:** Baseado no modelo **CARDIAC** (Bell Labs, anos 60).
- **Sistema Decimal:** Utiliza números de 6 dígitos com sinal em vez de binário.
- **Memória**: 100 endereços (00 a 99).
- **98**: Output (Impressora/Tela).
- **99**: Valor fixo 0 (utilizado para o comando HALT).

OISC Pedagógico (Drexel University)

A Instrução Única: s d t (SUBLEQ):

- A instrução é um bloco único de 6 dígitos: **SS DD TT.**
- **Cálculo**: Mem[d] = Mem[d] - Mem[s].
- **Lógica**: O processador verifica se o resultado em d é negativo (< 0).
- **Desvio**: Se for negativo, pula para o endereço t. Caso contrário, segue para a próxima instrução.

OISC Pedagógico (Drexel University)

Exemplo Prático: Realizando uma Soma (A + B):

- **Endereço 90: 000010 (Valor A)**
- **Endereço 91: 000023 (Valor B)**
- **Endereço 92: 000000 (Temporário 1)**
- **Endereço 93: 000000 (Temporário 2)**
- **Endereço 99: 000000 (Halt/Zero fixo)**

https://www.cs.drexel.edu/~bls96/oisc/OISC.html

Endereço	Instrução	Lógica Matemática	Descrição
00	**90 92 01**	Mem[92] = 0 - 10	Inverte o valor de A e salva em 92 (Temp). Agora 92 tem -10.
01	**92 91 02**	Mem[91] = 23 - (-10)	Subtrai o negativo, resultando em soma. Agora 91 tem 33.
02	**91 93 03**	Mem[93] = 0 - 33	Inverte o resultado (33) para -33 na célula 93..
03	**93 98 99**	Mem[98] = 0 - (-33)	Output: Subtrai o negativo do Output. O resultado final em 98 é 33.

O FlipJump

- É um exemplo de máquina de manipulação de bits, sendo a mais simples implementação de um OISC.
- **Modo de operação:**
  - Dados A e B, a instrução A;B inverte o valor do A-Ésimo bit e pula para a instrução no endereço B.
  - Tanto A quanto B podem ser vazios, ou seja, a instrução pode não inverter nenhum bit ou não realizar pulos (desvios)

```
1000;256  // addresses 000-127
32;446    // addresses 128-255
128;256   // addresses 256-383
```

Exemplo de código simples em FlipJump para um processador de 64 Bits

Transport Triggered Machines (TTA)

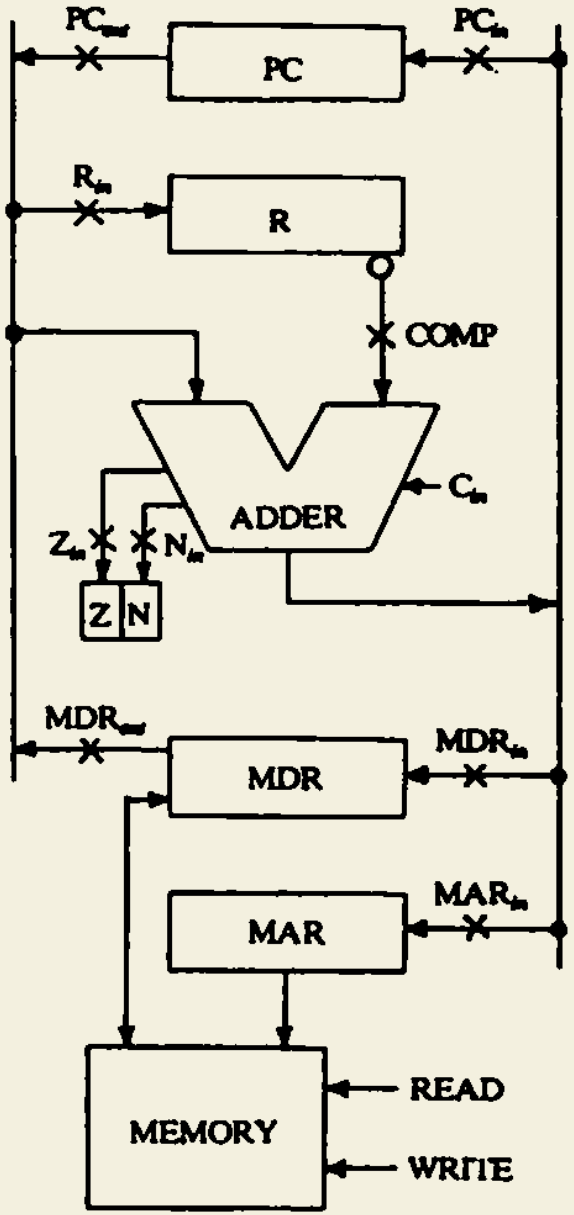
- Utilizam a movimentação entre registradores, junto ao uso de registradores especiais para realizar operações.

- Possuem como vantagem o fácil paralelismo de tarefas, o menor consumo de energia, a simplicidade de funcionamento e a alta capacidade de personalização.

- Utilizado em aplicações embarcadas de alto desempenho, redes neurais e processamenro de sinais.

URISC

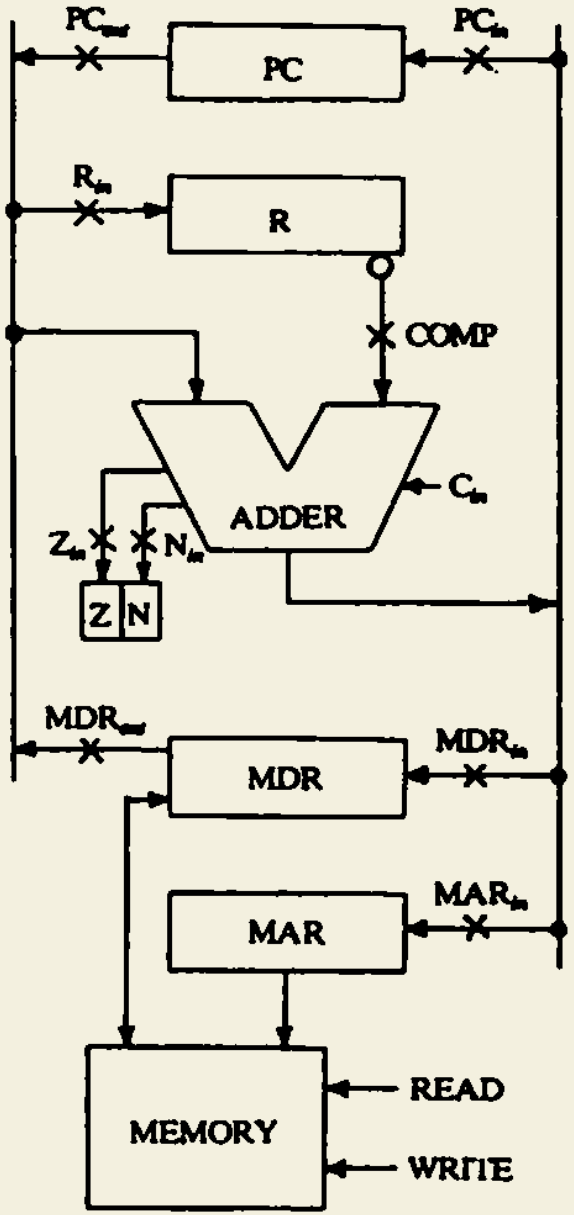
- Processador de 16 bits
- Memória de 64K x 16
- Barramentos de entrada e saída
- Instrução com 3 palavras de 16 bits



Datapath do URISC

Design do Datapath

- Barramentos
- ULA
- Registrador R
- Flags Z (Zero) e N (Negative)
- Memória principal
- Memory Address Register (MAR)
- Memory Data Register (MDR)
- Contador de Programa (PC)
- Sinais de controle



Datapath do URISC

A Instrução

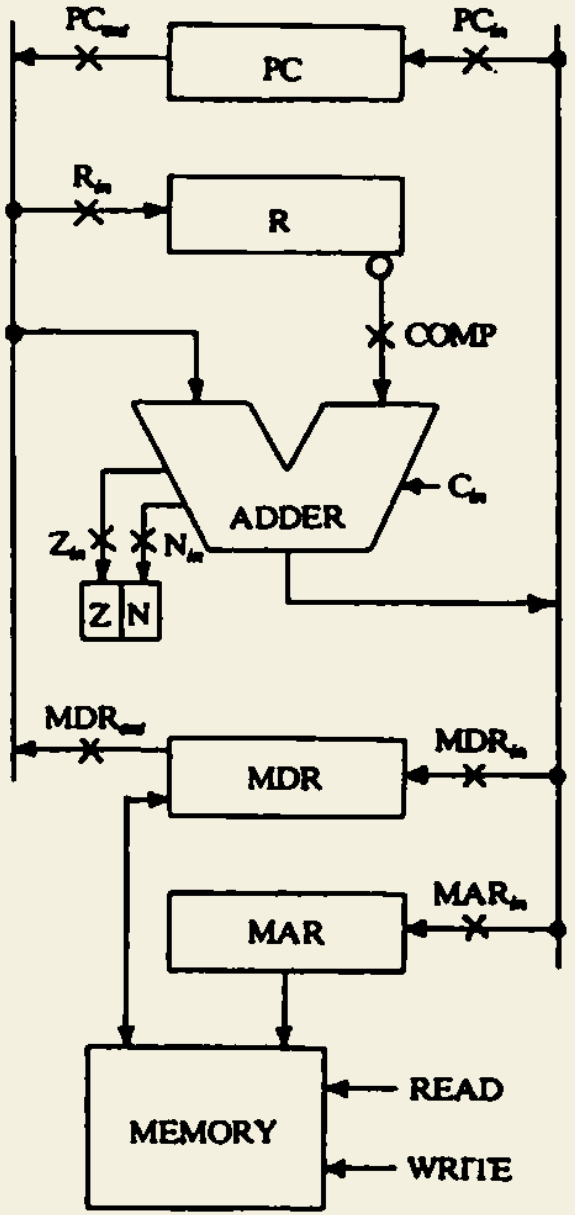
- A instrução ocupa 3 espaços consecutivos na memória

Operando A	Operando B	Endereço do Jump

- O processador executa uma sequência de passos para processar cada palavra da instrução.

Ciclo de Execução (9 Passos)

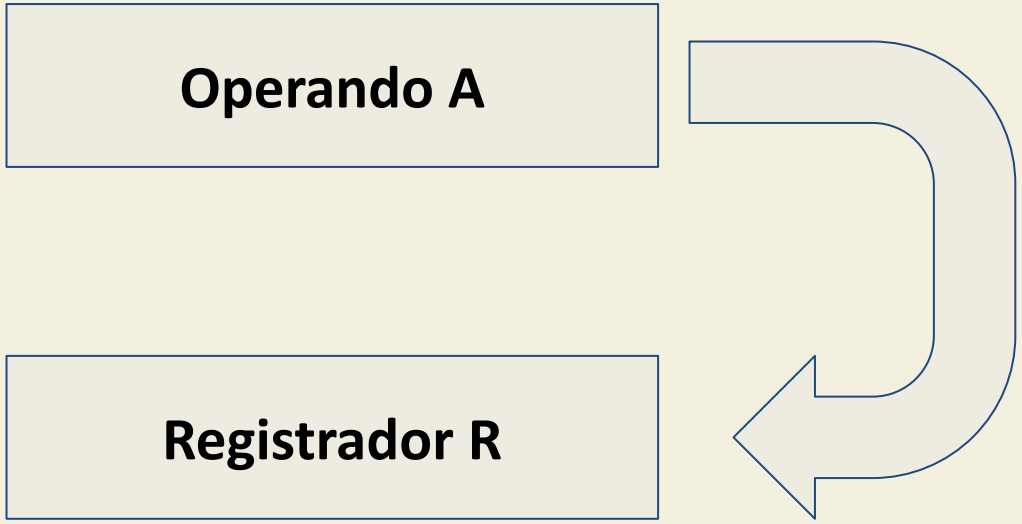
0. PCout, Zin, MARin, READ, ZEND
1. MDRout, MARin, READ
2. MDRout, Rin
3. PCout, Cin, PCin, MARin, READ
4. MDRout, MARin, READ
5. MDRout, COMP, Cin, Nin, MDRin, WRITE
6. PCout, Cin, PCin, MARin, READ
7. PCout, Cin, PCin, NNEND
8. MDRout, PCin

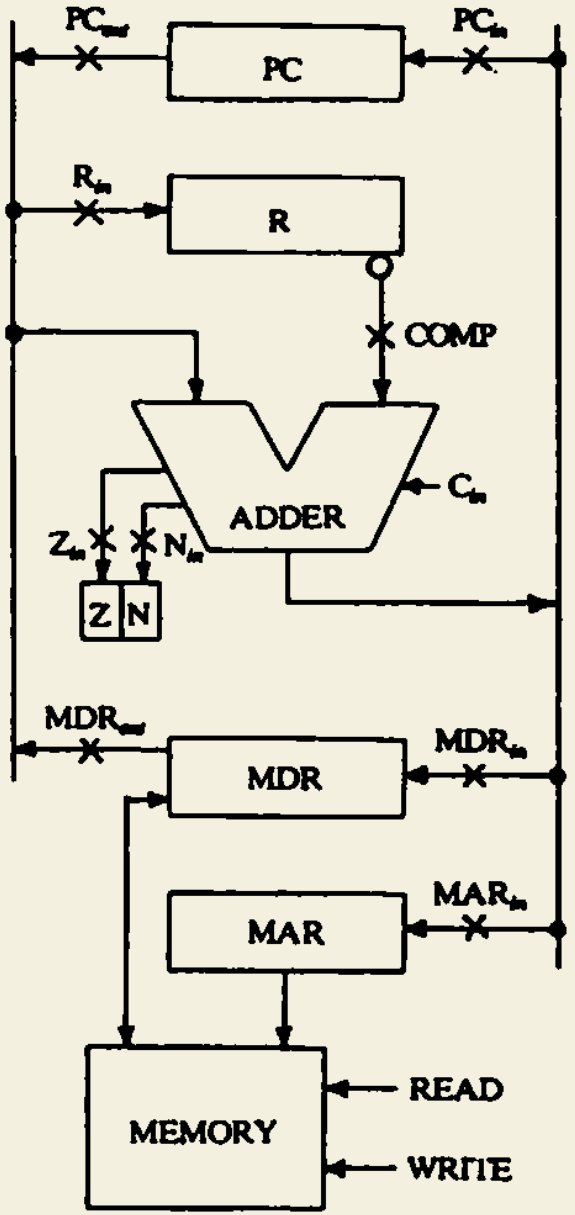


Datapath do URISC

Ciclo de Execução (9 Passos)

0. PCout, Zin, MARin, READ, ZEND
1. MDRout, MARin, READ
2. MDRout, Rin

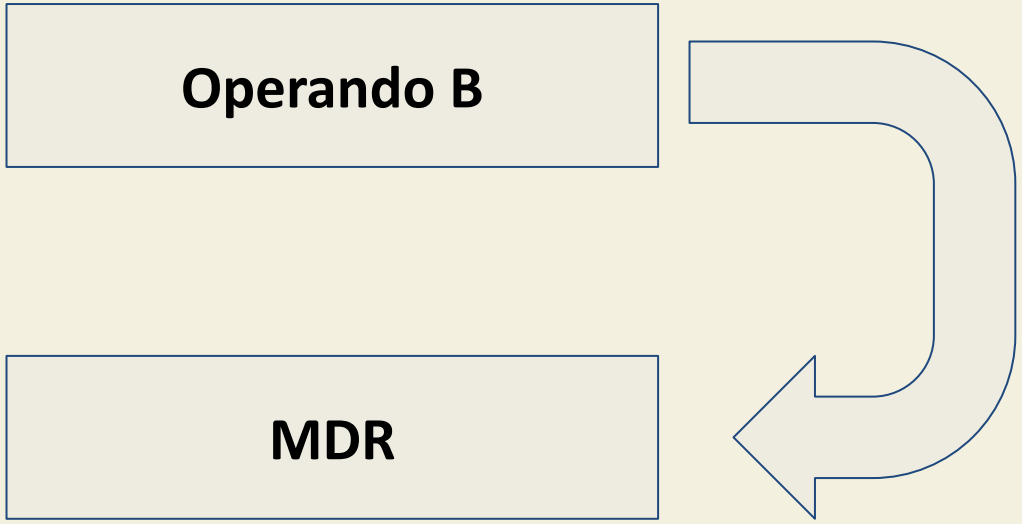


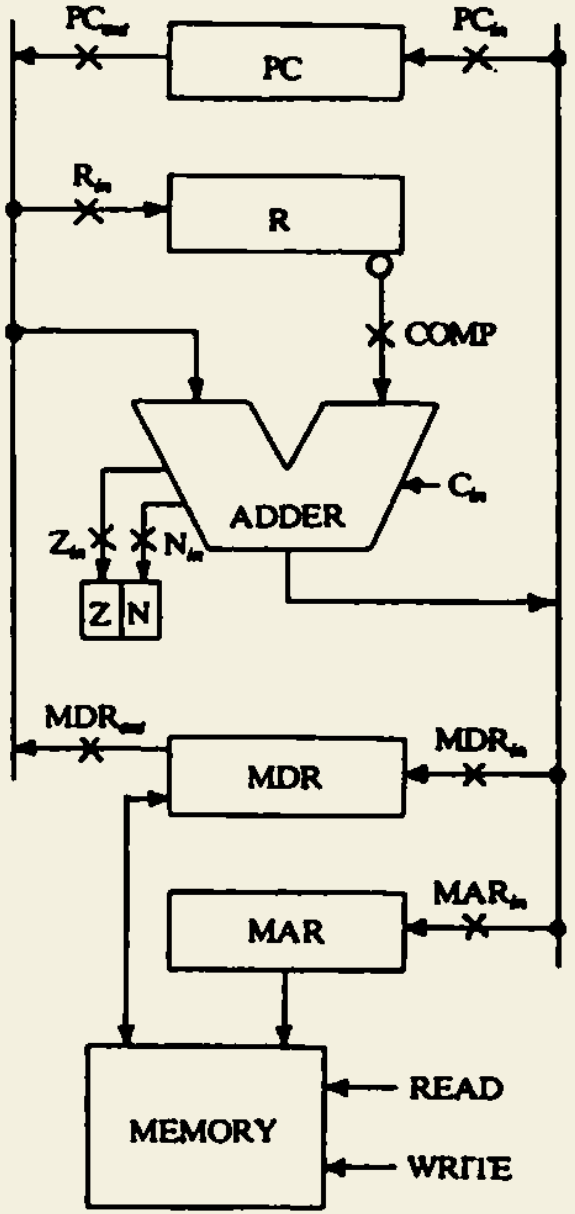


Datapath do URISC

Ciclo de Execução (9 Passos)

3. PCout, Cin, PCin, MARin, READ
4. MDRout, MARin, READ

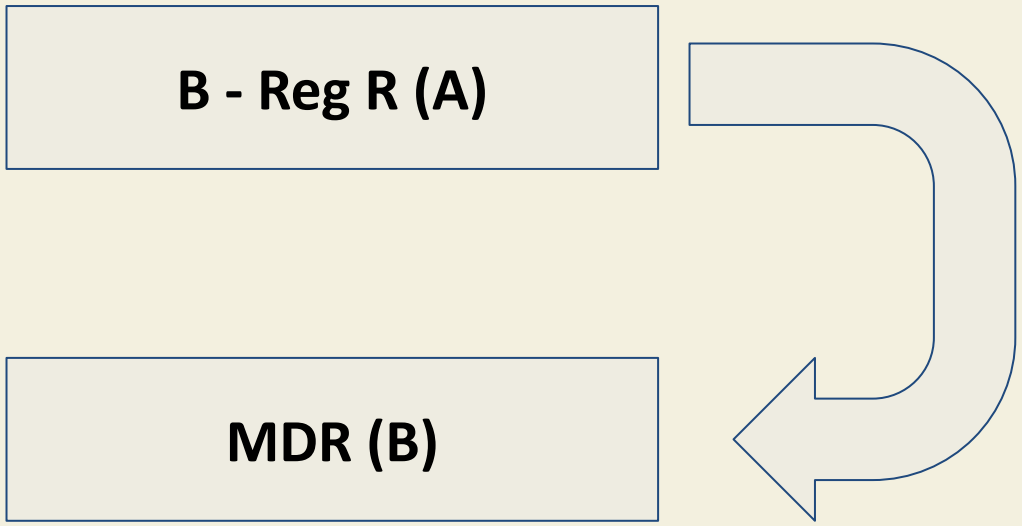


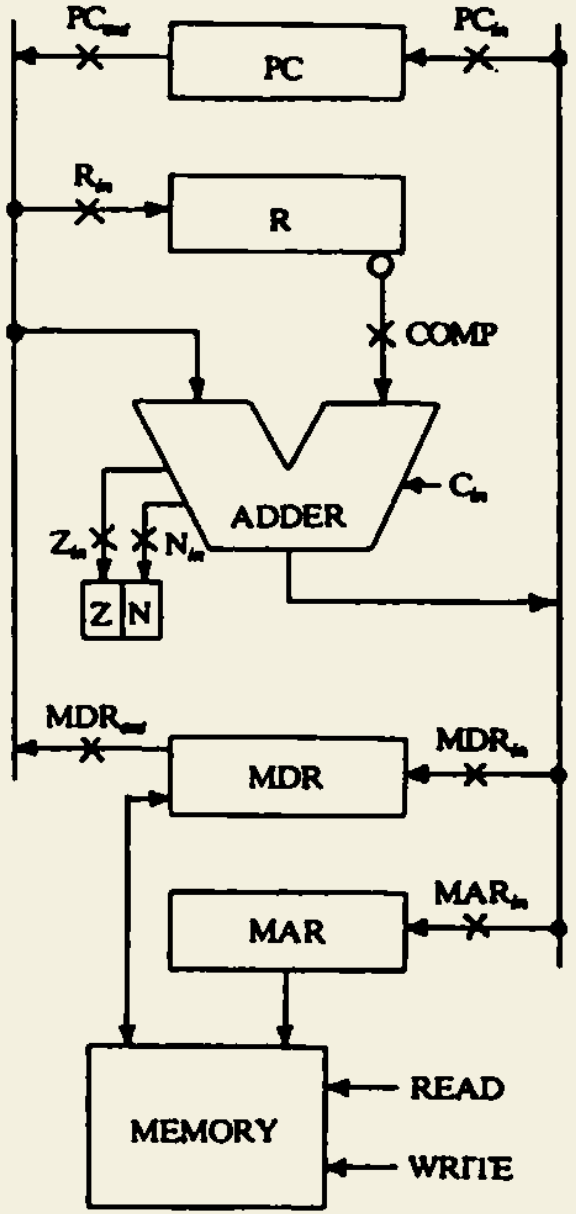


Datapath do URISC

Ciclo de Execução (9 Passos)

5. MDRout, COMP, Cin, Nin, MDRin, WRITE

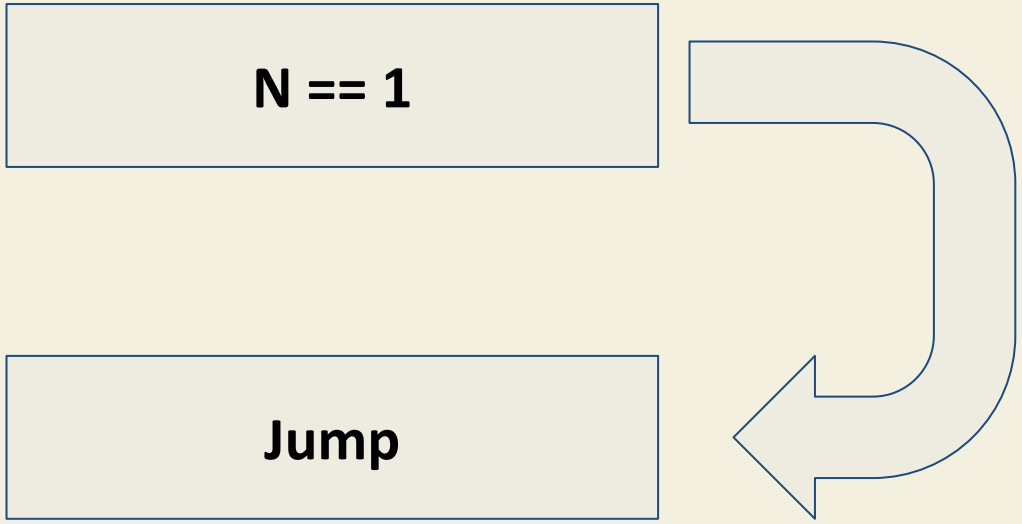


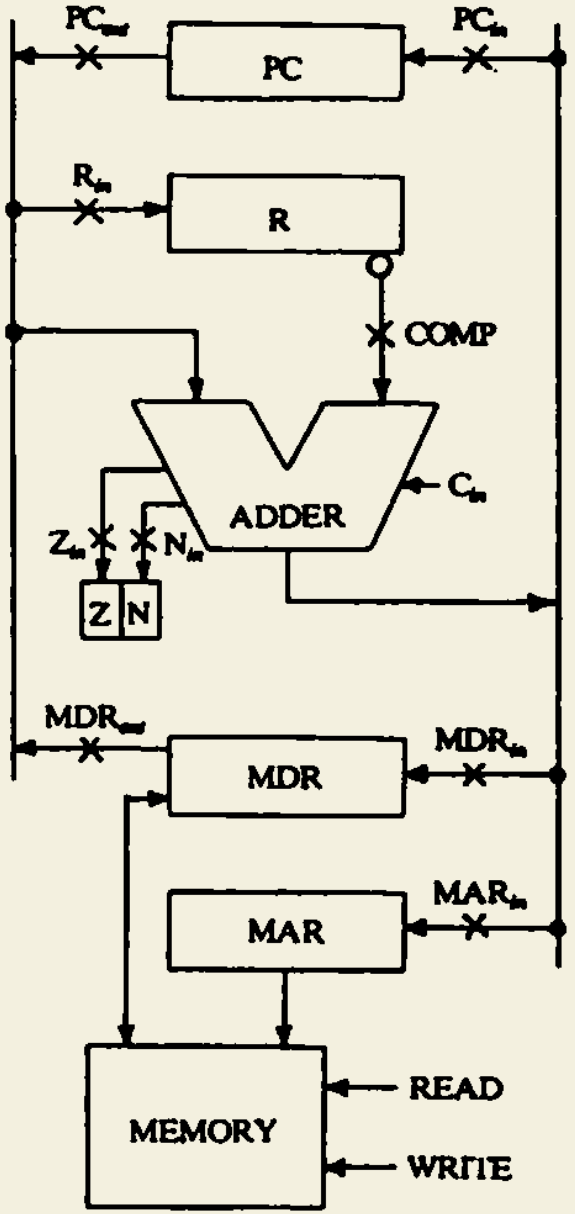


Datapath do URISC

Ciclo de Execução (9 Passos)

6. PCout, Cin, PCin, MARin, READ
7. PCout, Cin, PCin, NNEND
8. MDRout, PCin





Datapath do URISC

Microprogramado
X
Hardwired

Hardwired

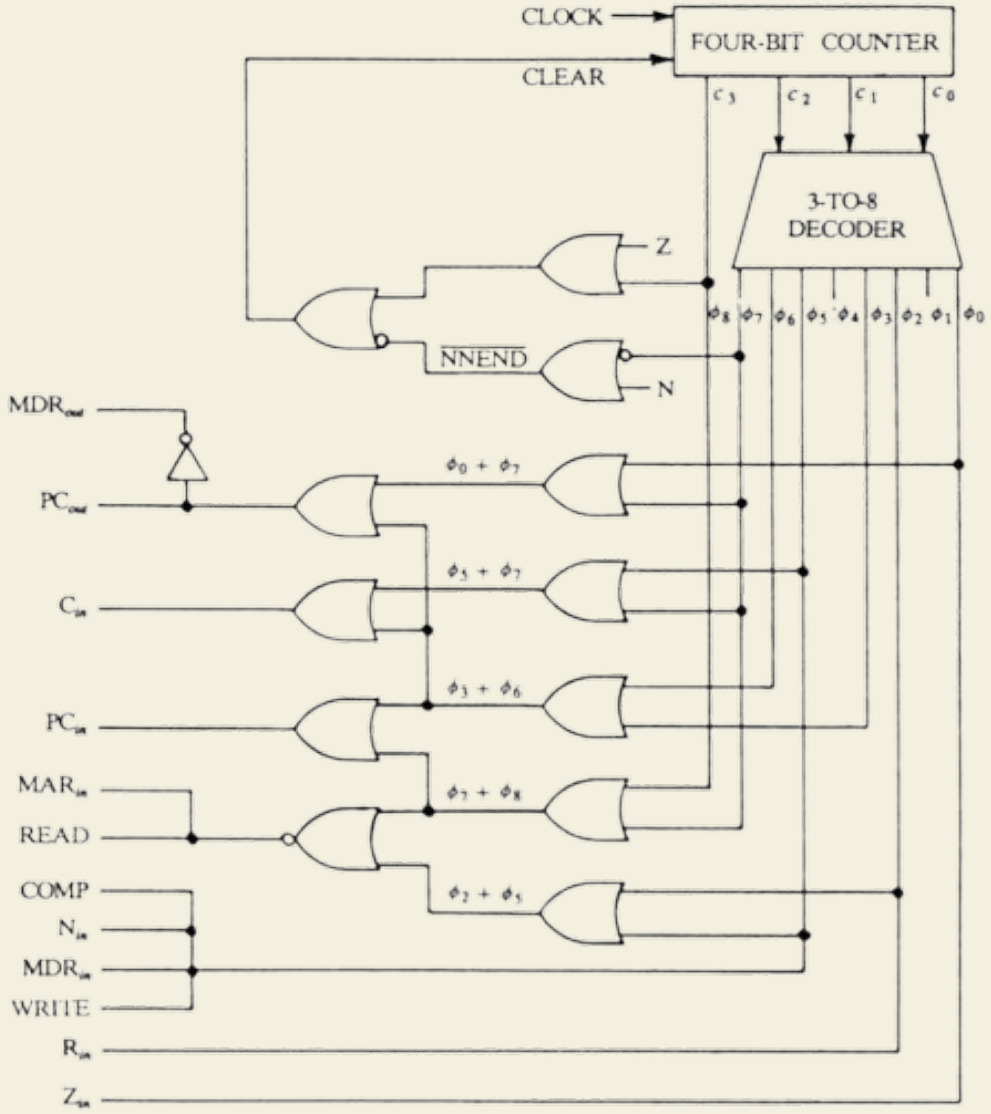


FIG. 2 Hardwired controller design for URISC.

$PC_{in} = \Phi_3 + \Phi_6 + \Phi_7 + \Phi_8$

$PC_{out} = \Phi_0 + \Phi_3 + \Phi_6 + \Phi_7$

$R_{in} = \Phi_2$

$COMP = \Phi_5$

$C_{in} = \Phi_3 + \Phi_5 + \Phi_6 + \Phi_7$

$Z_{in} = \Phi_0$

$N_{in} = \Phi_5$

$MDR_{in} = \Phi_5$

$MDR_{out} = \Phi_1 + \Phi_2 + \Phi_4 + \Phi_5 + \Phi_8$

$MAR_{in} = \Phi_0 + \Phi_1 + \Phi_3 + \Phi_4 + \Phi_6$

$READ = \Phi_0 + \Phi_1 + \Phi_3 + \Phi_4 + \Phi_6$

$WRITE = \Phi_5$

$ZEND = \Phi_0 Z$

$NNEND = \Phi_7 \overline{N}$

Otimização:

$COMP = N_{in} = MDR_{in} = WRITE$

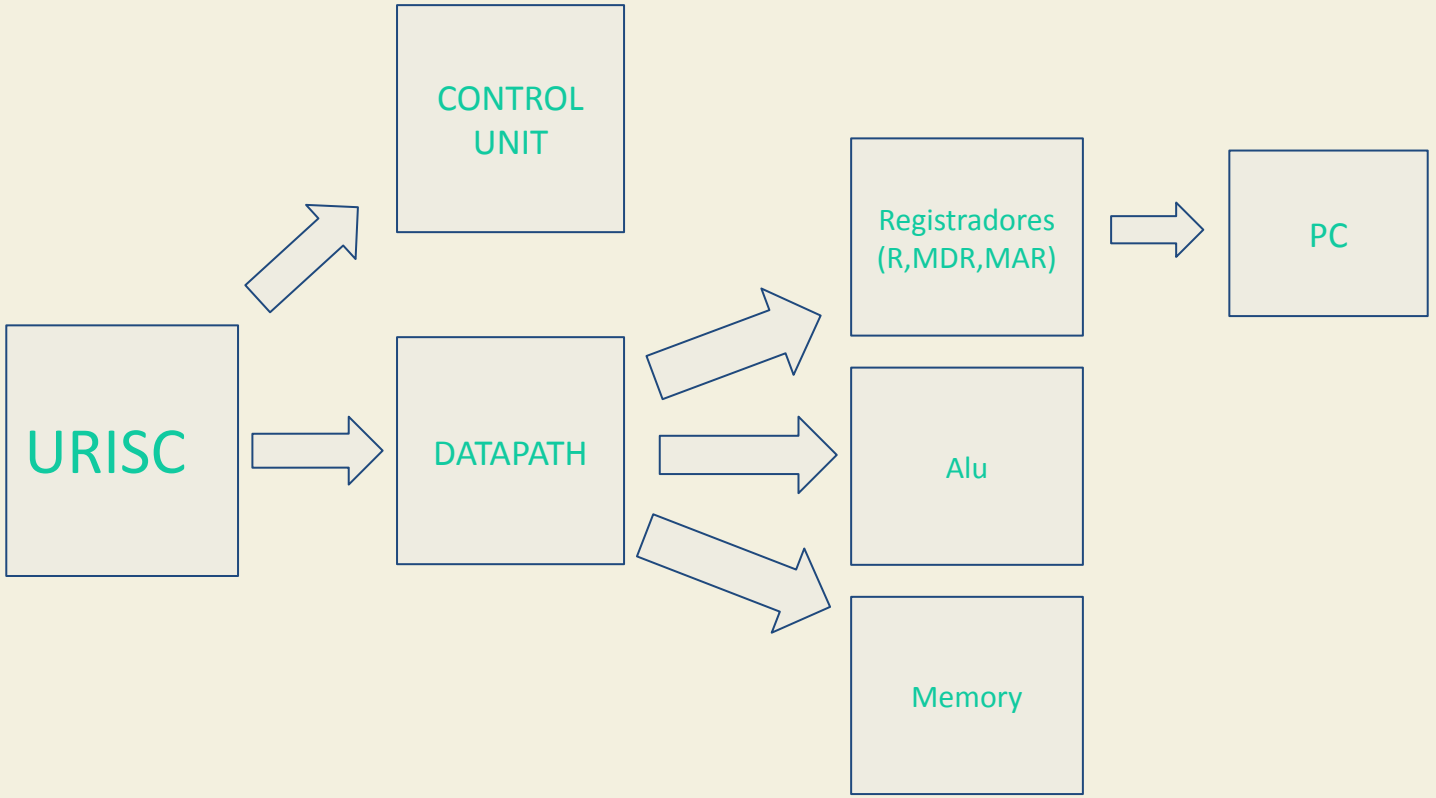
$MAR_{in} = READ$

$MDR_{out} = \overline{PC}_{out}$

Característica	Unidade de Controle Microprogramada	Unidade de Controle Hardwired
Mecanismo Central	Memória de Controle (ROM) + sequenciador	Portas Lógicas + Contador + Decodificador
Definição da Lógica	Algorítmica/Software (Microcódigo)	Estrutural/Hardware (Fiação/Gates)
Flexibilidade de Design	Alta: alterações exigem apenas reprogramação da ROM.	Baixa: alterações exigem reconfiguração física das portas.
Complexidade Aparente	Moderada: Requer infraestrutura de memória e busca.	Baixa: Surpreendentemente, no URISC é simples e direta.
Área em Silício (VLSI)	Maior: overhead da ROM e decodificadores.	Muito pequena: Ideal para integração em larga escala.
Velocidade	Limitada pelo tempo de acesso à memória de controle.	Limitada apenas pelo atraso de propagação das portas.
Natureza Didática	Ideal para ensinar o fluxo e a lógica sequencial.	Ideal para desmistificar a construção física do computador.

Implementção Verilog do URISC

Descrição Top-Down



URISC (Top-Level):

URISC.v — Visual Studio Code

```
module Processor (
    input wire clk,
    input wire reset
);
    // --- Barramento de Controle (O "Sistema Nervoso") ---
    // Estes fios transportam os comandos da Unidade de Controle para ativar/desativar
    // componentes específicos no Datapath (ex: abrir a porta do PC, gravar na Memória).
    wire ctrl_pc_out, ctrl_pc_in, ctrl_pc_inc;
    wire ctrl_r_in, ctrl_mar_in;
    wire ctrl_mdr_in, ctrl_mdr_out;
    wire ctrl_read_mem, ctrl_write_mem;
    wire ctrl_comp_alu;

    // Sinal crítico para arquiteturas sequenciais:
    // Como a decisão de pulo (Branch) ocorre ciclos DEPOIS da subtração,
    // precisamos congelar (salvar) o resultado das Flags Z e N.
    wire ctrl_save_flags;

    // --- Sinais de Status (Feedback) ---
    // O Datapath informa ao Controle o resultado da última operação matemática.
    wire status_z;
    wire status_n;
```

URISC (Top-Level):

URISC.v — Visual Studio Code

```
// --- 1. Instância do Cérebro (Unidade de Controle) ---
// Responsável por orquestrar a sequência de micro-operações.
// Em URISC, como só há uma instrução, não há "Decodificação de Opcode",
// apenas uma máquina de estados fixa.
ControlUnit controle (
    .clk(clk),
    .reset(reset),
    // Entradas: Toma decisões baseadas no estado atual do processador (Flags)
    .flag_z(status_z),
    .flag_n(status_n),
    // Saídas: Comanda os componentes
    .pc_out(ctrl_pc_out), .pc_in(ctrl_pc_in), .pc_inc(ctrl_pc_inc),
    .r_in(ctrl_r_in), .mar_in(ctrl_mar_in),
    .mdr_in(ctrl_mdr_in), .mdr_out(ctrl_mdr_out),
    .read_mem(ctrl_read_mem), .write_mem(ctrl_write_mem),
    .comp_alu(ctrl_comp_alu),

    .save_flags(ctrl_save_flags)
);
```

URISC (Top-Level):

URISC.v — Visual Studio Code

```
43
44      // --- 2. Instância do Corpo (Datapath) ---
45      // Onde os dados realmente fluem e são processados.
46      // Contém os Registradores, a ALU e a Memória.
47      Datapath caminho (
48          .clk(clk),
49          .reset(reset),
50          // Recebe ordens cegas do Controle
51          .pc_out(ctrl_pc_out), .pc_in(ctrl_pc_in), .pc_inc(ctrl_pc_inc),
52          .r_in(ctrl_r_in), .mar_in(ctrl_mar_in),
53          .mdr_in(ctrl_mdr_in), .mdr_out(ctrl_mdr_out),
54          .read_mem(ctrl_read_mem), .write_mem(ctrl_write_mem),
55          .comp_alu(ctrl_comp_alu),
56
57          .save_flags(ctrl_save_flags),
58
59          // Envia o status do processamento
60          .flag_z(status_z),
61          .flag_n(status_n)
62      );
63  endmodule
```

DATAPATH:

URISC.v — Visual Studio Code

```
module Datapath (
    input  wire clk,
    input  wire reset,

    // Sinais de Controle
    input  wire pc_in, pc_out,
    input  wire pc_inc,
    input  wire r_in,
    input  wire mar_in,
    input  wire mdr_in, mdr_out,
    input  wire read_mem, write_mem,
    input  wire comp_alu,
    input  wire save_flags,

    // Saídas para a Unidade de Controle
    output reg flag_z,
    output reg flag_n
);

    wire [15:0] bus;
    wire [15:0] pc_val;
    wire [15:0] r_val;
    wire [15:0] mar_val;
    wire [15:0] mdr_val_out;
    wire [15:0] mem_data_out;
    wire [15:0] alu_result;
    wire [15:0] mdr_data_in;

    // Fios temporários para a saída instantânea da ALU
    wire alu_z_instantaneo;
    wire alu_n_instantaneo;

```

DATAPATH:

URISC.v — Visual Studio Code

```
32
33      assign bus = (pc_out)  ? pc_val :
34                   (mdr_out) ? mdr_val_out :
35                   16'd0;
36
37      // --- Lógica de Salvamento das Flags ---
38      always @(posedge clk or posedge reset) begin
39          if (reset) begin
40              flag_z <= 0;
41              flag_n <= 0;
42          end else if (save_flags) begin
43              // Tira uma "foto" das flags neste exato momento
44              flag_z <= alu_z_instantaneo;
45              flag_n <= alu_n_instantaneo;
46          end
47      end
48
49      // --- Componentes ---
50      Register #(.WIDTH(16), .HAS_INC(1)) PC (.clk(clk), .reset(reset), .load(pc_in), .inc(pc_inc), .data_in(bus), .data_out(pc_val));
51      Register #(.WIDTH(16), .HAS_INC(0)) R (.clk(clk), .reset(reset), .load(r_in), .inc(1'b0), .data_in(bus), .data_out(r_val));
52      Register #(.WIDTH(16), .HAS_INC(0)) MAR (.clk(clk), .reset(reset), .load(mar_in), .inc(1'b0), .data_in(bus), .data_out(mar_val));
53
54      assign mdr_data_in = (read_mem) ? mem_data_out : (comp_alu) ? alu_result : bus;
55      wire mdr_load_enable = mdr_in | read_mem;
56      Register #(.WIDTH(16)) MDR (.clk(clk), .reset(reset), .load(mdr_load_enable), .data_in(mdr_data_in), .data_out(mdr_val_out));
57
58      Memory RAM (.clk(clk), .addr(mar_val), .data_in(mdr_val_out), .write_en(write_mem), .read_en(read_mem), .data_out(mem_data_out));
59
```

DATAPATH:

URISC.v — Visual Studio Code

```
59
60      // --- ALU Conectada aos Fios Temporários ---
61      ALU alu_instance (
62          .bus_in(bus),
63          .r_in(r_val),
64          .comp(comp_alu),
65          .result(alu_result),
66          .flag_z(alu_z_instantaneo),
67          .flag_n(alu_n_instantaneo)
68      );
69
70  endmodule
```

ALU(ADDER):

URISC.v — Visual Studio Code

```v
module ALU (
    input  wire [15:0] bus_in,  // Operando B (Vem do Barramento)
    input  wire [15:0] r_in,    // Operando A (Vem do Registrador R)
    input  wire comp,           // Sinal de Controle: 0=Soma, 1=Subtração
    output wire [15:0] result,  // Resultado
    output wire flag_n,         // Flag Negativo (Bit mais significativo)
    output wire flag_z          // Flag Zero (Nor de todos os bits)
);

    // --- Lógica do Complemento de 2 ---
    // Para subtrair A de B (B - A), o hardware faz: B + (~A) + 1

    // 1. Inversão (NOT):
    // Se o sinal 'comp' estiver ativo, invertemos todos os bits de A.
    wire [15:0] r_ajustado;
    assign r_ajustado = (comp) ? ~r_in : r_in;

    // 2. Somador (Adder):
    // Somamos B + A_invertido.
    // O trunfo: O sinal 'comp' (que é 1 na subtração) é somado na entrada de Carry In.
    // Isso completa a lógica do "+1" do Complemento de 2.
    assign result = bus_in + r_ajustado + comp;

    // 3. Geração de Flags:
    // Estas flags são geradas combinacionalmente (instantaneamente).
    assign flag_z = (result == 16'd0);
    assign flag_n = result[15]; // Em complemento de 2, o bit 15 é o sinal.

endmodule
```

Registradores:

URISC.v — Visual Studio Code

```
module Register #(
    parameter WIDTH = 16,
    parameter HAS_INC = 0 // 0 = Registrador Comum, 1 = Registrador com Soma
)(
    input  wire clk,
    input  wire reset,
    input  wire load,       // Carga paralela (ex: Pulo/Branch)
    input  wire inc,
    input  wire [WIDTH-1:0] data_in,
    output reg  [WIDTH-1:0] data_out
);

    always @(posedge clk or posedge reset) begin
        if (reset) begin
            data_out <= 0;
        end else if (load) begin
            data_out <= data_in;
        end else if (HAS_INC && inc) begin
            // Só compila essa lógica se HAS_INC = 1
            data_out <= data_out + 1;
        end
    end

endmodule
```

Memory:

URISC.v — Visual Studio Code

```
module Memory(
    input  wire clk,
    input  wire [15:0] addr,     // Endereço de 16 bits (Vem do MAR)
    input  wire [15:0] data_in,  // 16 bits de dados (Vem do MDR)
    input  wire write_en,
    input  wire read_en,
    output reg  [15:0] data_out  //16 bits de dados (Vai para o MDR)
);


    reg [15:0] ram_block [0:(1<<10)-1];

    // 2. Escrita
    always @(posedge clk) begin
        if (write_en) begin

            ram_block[addr[9:0]] <= data_in;
        end
    end

    // 3. Leitura
    always @(*) begin
        if (read_en)
            data_out = ram_block[addr[9:0]];
        else
            data_out = 16'd0; // Zera saída
    end

    // 4. Inicialização
    initial begin
        $readmemh("programa.hex", ram_block);
    end

endmodule
```

Control Unit:

URISC.v — Visual Studio Code

```
1  module ControlUnit (
2      input  wire clk,
3      input  wire reset,
4      input  wire flag_z,
5      input  wire flag_n,
6
7      output reg pc_out, pc_in, pc_inc,
8      output reg r_in,
9      output reg mar_in,
10     output reg mdr_in, mdr_out,
11     output reg read_mem, write_mem,
12     output reg comp_alu,
13     output reg save_flags
14 );
```

Control Unit:

URISC.v — Visual Studio Code

```
16      integer estado_atual;
17
18      // --- Máquina de Estados Finitos (FSM) ---
19      // Define a coreografia dos dados a cada pulso de clock.
20      always @(posedge clk or posedge reset) begin
21          if (reset) estado_atual <= 0;
22          else begin
23              case (estado_atual)
24                  0: estado_atual <= 1; // Start
25
26                  // == FASE 1: BUSCA DO PRIMEIRO OPERANDO (A) ==
27                  // O URISC usa Indireção: O código contém o ENDEREÇO de A, não o valor.
28                  // 1-3: Buscamos o ponteiro "A" na memória de programa.
29                  // 4-5: Usamos esse ponteiro para buscar o DADO real na memória de dados.
30                  1: estado_atual <= 2; 2: estado_atual <= 3;
31                  3: estado_atual <= 4; 4: estado_atual <= 5; 5: estado_atual <= 6;
32
33                  // == FASE 2: BUSCA DO SEGUNDO OPERANDO (B) ==
34                  // Mesmo processo: Busca Ponteiro -> Configura Endereço -> Busca Dado.
35                  // Ao final (Estado 9), o MAR (Registrador de Endereço) aponta para B.
36                  // Isso é crucial para gravarmos o resultado de volta em B depois.
37                  6: estado_atual <= 7; 7: estado_atual <= 8; 8: estado_atual <= 9;
38                  9: estado_atual <= 10;
39
40                  // == FASE 3: EXECUÇÃO (ALU) ==
41                  // Realiza a subtração B - A.
42                  10: estado_atual <= 11;
43
44                  // == FASE 4: WRITEBACK (Escrita) ==
45                  // Salva o resultado da subtração na memória (sobrescreve B).
46                  11: estado_atual <= 12;
47
48                  // == FASE 5: BUSCA DO DESTINO (C) E BRANCH ==
49                  // Busca o terceiro operando (C), que é o endereço de pulo.
50                  // Decide se altera o PC (Pulo) ou apenas incrementa (Próxima instrução).
51                  12: estado_atual <= 13;
52                  13: estado_atual <= 14;
53                  14: estado_atual <= 1; // Reinicia o ciclo (Loop Infinito)
54                  default: estado_atual <= 0;
55              endcase
56          end
57      end
```

Control Unit:

URISC.v — Visual Studio Code

```
60  // Em cada estado, definimos quais "chaves" (tristates) do hardware ligar.
61  always @(*) begin
62      // Reset dos sinais (Segurança)
63      pc_out = 0; pc_in = 0; pc_inc = 0; r_in = 0; mar_in = 0;
64      mdr_in = 0; mdr_out = 0; read_mem = 0; write_mem = 0; comp_alu = 0;
65      save_flags = 0;
66
67      case (estado_atual)
68          // Lógica de Movimentação de Dados (Busca Indireta)
69          1: begin pc_out = 1; mar_in = 1; read_mem = 1; end // PC -> MAR
70          2: begin read_mem = 1; end                         // Espera Memória
71          3: begin mdr_out = 1; mar_in = 1; end              // Ponteiro A -> MAR
72          4: begin read_mem = 1; end                         // Lê Valor A
73          5: begin mdr_out = 1; r_in = 1; pc_inc = 1; end    // Valor A -> Reg R (Guarda A)
74
75          6: begin pc_out = 1; mar_in = 1; read_mem = 1; end // PC -> MAR (Buscando B)
76          7: begin read_mem = 1; end
77          8: begin mdr_out = 1; mar_in = 1; end              // Ponteiro B -> MAR
78          9: begin read_mem = 1; end                         // Lê Valor B
79
80          // --- PONTO CRÍTICO: A OPERAÇÃO ---
81          10: begin
82              mdr_out = 1;   // Coloca Valor B no Barramento
83              comp_alu = 1;  // Ativa Subtração na ALU (Bus - R)
84              mdr_in = 1;    // O resultado volta para o MDR
85              read_mem = 0;  // Desliga memória para evitar conflito no barramento
86
87              // Salva o estado da ALU (Zero/Negativo) neste exato momento.
88              // Se não salvarmos aqui, perderemos essa informação no próximo clock.
89              save_flags = 1;
90          end
91
92          // Escrita na Memória
93          11: begin write_mem = 1; pc_inc = 1; end // Grava MDR na pos B (MAR ainda aponta p/ B)
94
95          // Preparação para o Pulo
96          12: begin pc_out = 1; mar_in = 1; read_mem = 1; end // PC -> MAR (Endereço C)
97          13: begin read_mem = 1; end // Lê C
98
99          // --- PONTO CRÍTICO: DECISÃO (BRANCH) ---
100         14: begin
101             // Consulta as Flags que foram salvas no Estado 10.
102             if (flag_z || flag_n) begin
103                 mdr_out = 1; // Coloca endereço C no barramento
104                 pc_in = 1;   // Carga Paralela no PC (JUMP)
105             end else begin
106                 pc_inc = 1;  // Apenas avança PC (PC + 1)
```

Mas afinal... é Turing completo mesmo?

- Para provar a Turing Completude do Subleq utilizamos o método da redução
- Consiste em provar que o Subleq consegue fazer a mesma operação de outra máquina Turing completa
- Usaremos a Máquina de Minsky como exemplo.

Mas afinal... é Turing completo mesmo?

- A máquina de Minsky possui 3 características
- Capacidade de guardar números em variáveis
- Somar e subtrair 1
- Conseguir desviar o código se um valor for zero

Mas afinal... é Turing completo mesmo?

- A máquina de Minsky possui 3 características
- Capacidade de guardar números em variáveis
- Somar e subtrair 1
- Conseguir desviar o código se um valor for zero

O Subleq faz tudo isso !

A Multi-One Instruction Set Computer for Microcontroller Applications

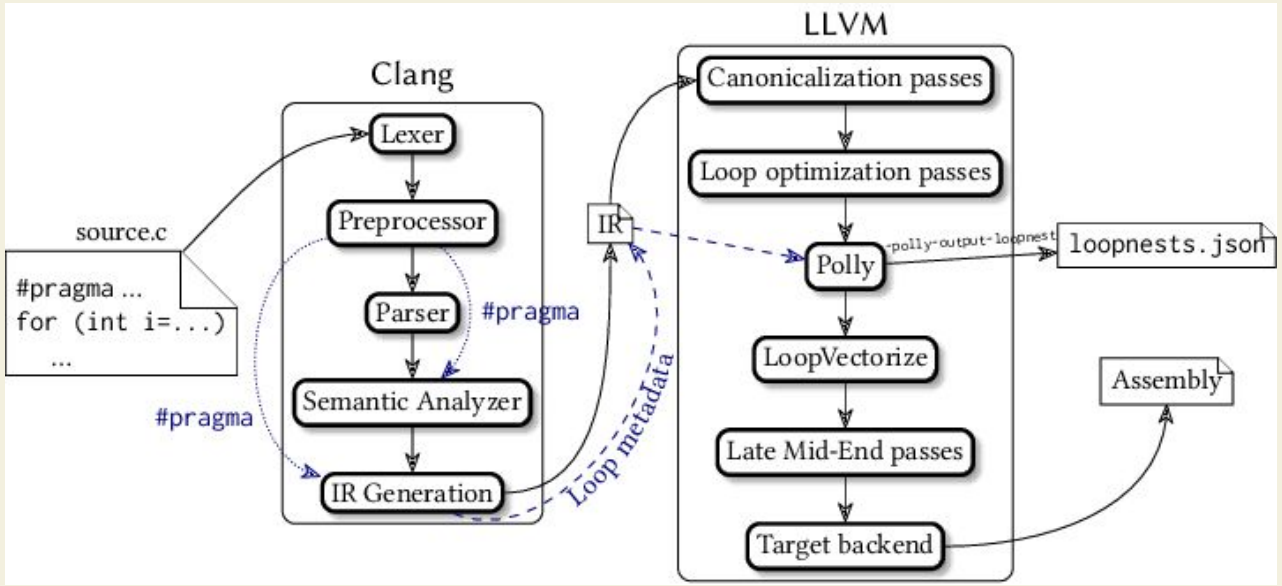
MARCO CREPALDI ANDREA MERELLO e MIRCO DI SALVO.
Publicado em 11 de agosto de 2021.

- Motivações
- Introdução e fundamentação teórica
- Pipeline de compilação

Multi Modo

refere-se a uma arquitetura capaz de suportar e executar diferentes modos de instrução ou conjuntos de instruções distintos no mesmo hardware.

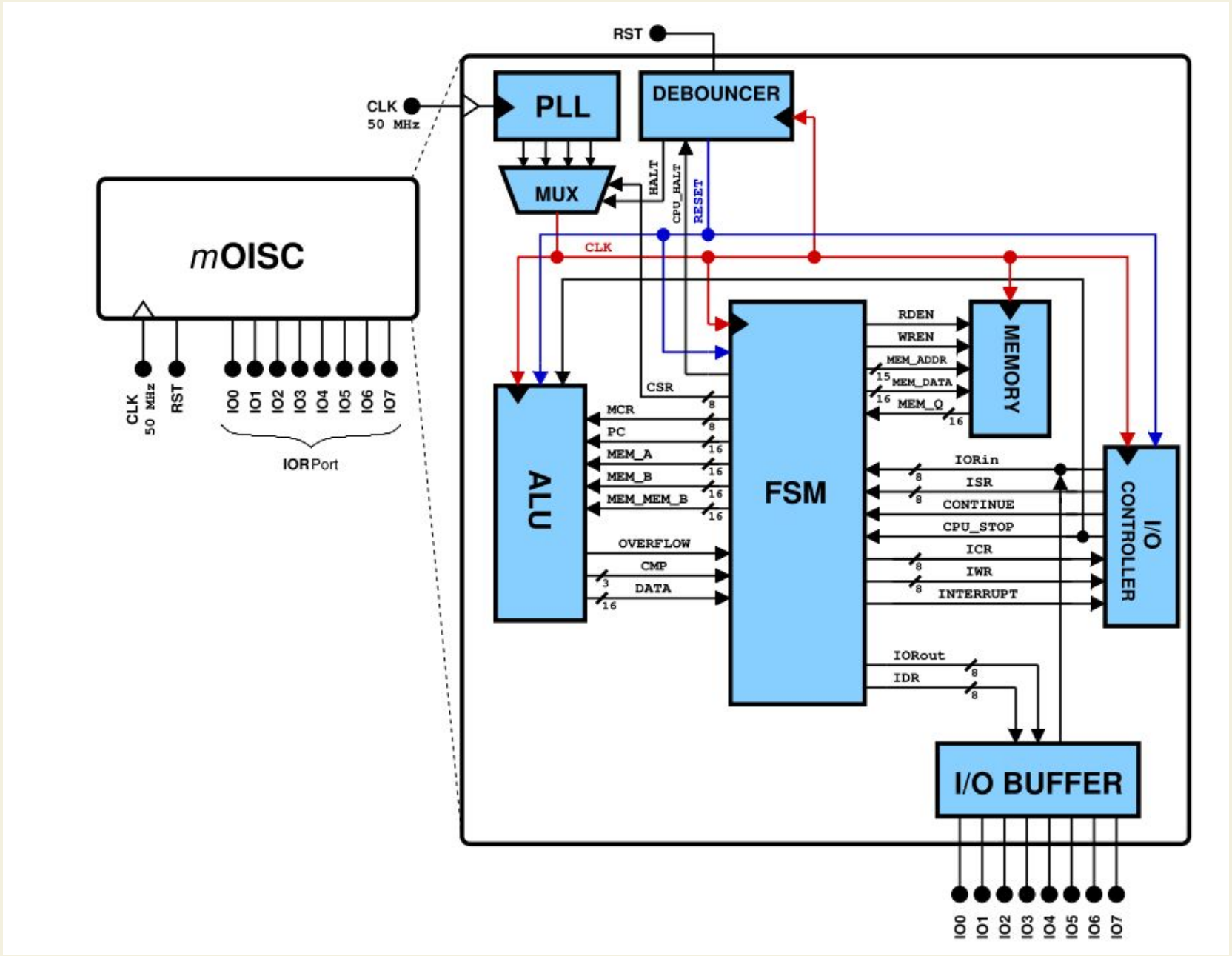
Compilação Modular

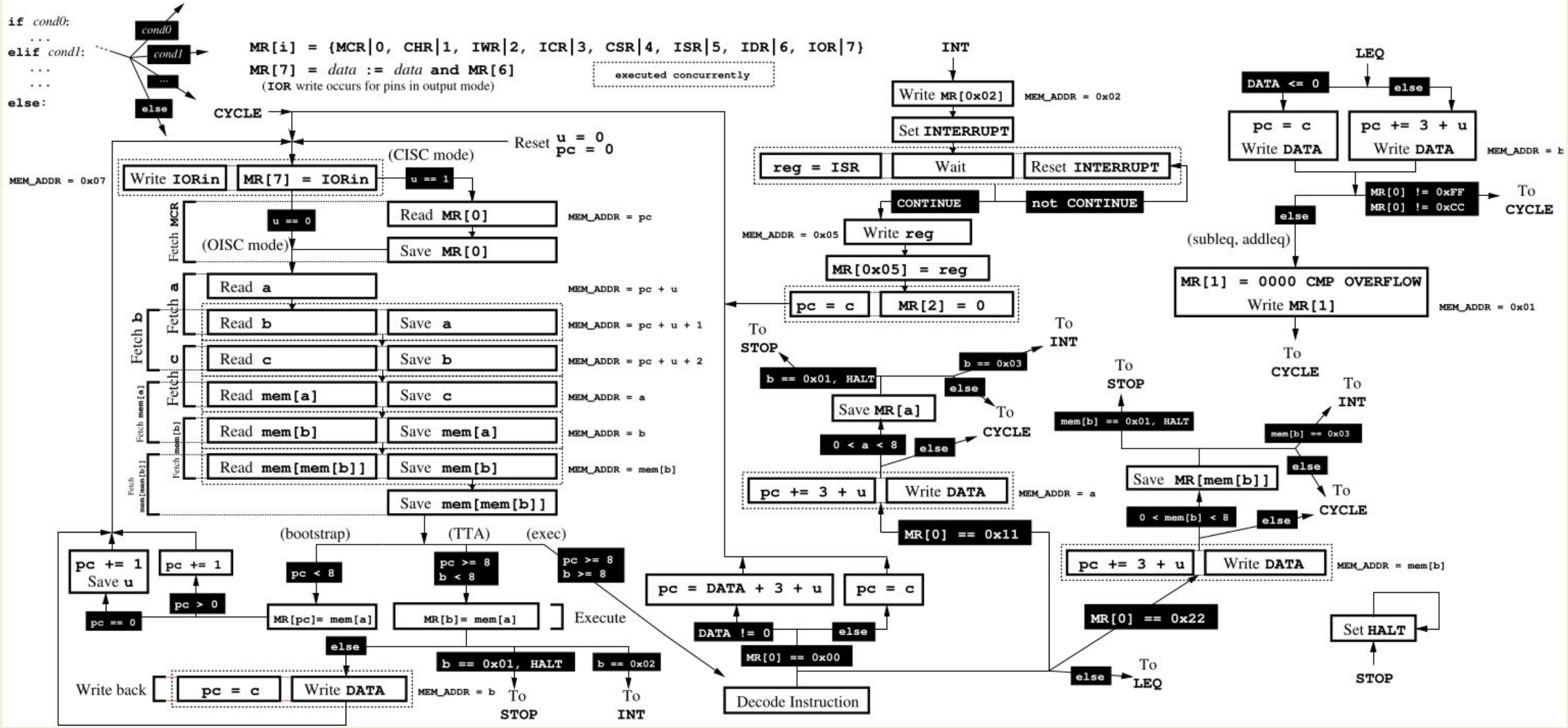


- **Compatibilidade do LLVM-IR com as arquiteturas mais utilizadas**

Organização dos registradores no mOISC

REGISTRADORES	FUNÇÃO
MCR (0x00)	Define qual operação OISC será executada
CHR (0x01)	Provê as flags de comparação e de overflow e para a CPU se o valor de 0xFF for colocado nele
IWR (0x02)	Responsável por interromper indefinitivamente o funcionamento da CPU, os valores colocados neles estão relacionados aos I/O Pin
ICR (0x03)	Define a direção de transição dos I/O Pins, ou seja, se funcionam em uma borda de subida ou descida
CSR (0x04)	Define a frequência do clock da CPU
ISR (0x05)	Mostra qual pino disparou o sinal para a CPU voltar a trabalhar
IDR (0x06)	Diz se os I/O Pin serão de input (bit 0) ou output (bit 1)
IOR (0x07)	Por onde ocorre a comunicação com o mundo externo


RST
CLK
50 MHz
PLL
DEBOUNCER
MUX
HALT
CPU_HALT
RESET
CLK
mOISC
CLK
50 MHz
RST
IO0
IO1
IO2
IO3
IO4
IO5
IO6
IO7
IOR Port
ALU
FSM
MEMORY
I/O CONTROLLER
I/O BUFFER
CSR
MCR
PC
MEM_A
MEM_B
MEM_MEM_B
OVERFLOW
CMP
DATA
RDEN
WREN
MEM_ADDR
MEM_DATA
MEM_Q
IORin
ISR
CONTINUE
CPU_STOP
ICR
IWR
INTERRUPT
IORout
IDR


if cond0:
elif cond1:
else:
cond0
cond1
else
MR[i] = {MCR|0, CHR|1, IWR|2, ICR|3, CSR|4, ISR|5, IDR|6, IOR|7}
MR[7] = data := data and MR[6]
(IOR write occurs for pins in output mode)
executed concurrently
CYCLE
Reset u = 0 pc = 0
(CISC mode)
Write IORin
MR[7] = IORin
MEM_ADDR = 0x07
u == 1
u == 0
(OISC mode)
Fetch MCR
Read MR[0]
Save MR[0]
MEM_ADDR = pc
Fetch a
Read a
MEM_ADDR = pc + u
Fetch b
Read b
Save a
MEM_ADDR = pc + u + 1
Fetch c
Read c
Save b
MEM_ADDR = pc + u + 2
Fetch mem[a]
Read mem[a]
Save c
MEM_ADDR = a
Fetch mem[b]
Read mem[b]
Save mem[a]
MEM_ADDR = b
Fetch mem[mem[b]]
Read mem[mem[b]]
Save mem[b]
MEM_ADDR = mem[b]
Save mem[mem[b]]
(bootstrap)
(TTA)
(exec)
pc < 8
pc >= 8
b < 8
pc >= 8
b >= 8
pc += 1
Save u
pc += 1
pc == 0
pc > 0
MR[pc]= mem[a]
MR[b]= mem[a]
Execute
else
Write back
pc = c
Write DATA
MEM_ADDR = b
b == 0x01, HALT
To STOP
b == 0x02
To INT
pc = DATA + 3 + u
pc = c
DATA != 0
else
MR[0] == 0x00
Decode Instruction
INT
Write MR[0x02]
MEM_ADDR = 0x02
Set INTERRUPT
reg = ISR
Wait
Reset INTERRUPT
CONTINUE
not CONTINUE
MEM_ADDR = 0x05
Write reg
MR[0x05] = reg
pc = c
MR[2] = 0
To STOP
b == 0x01, HALT
b == 0x03
To INT
else
To CYCLE
Save MR[a]
0 < a < 8
else
To CYCLE
pc += 3 + u
Write DATA
MEM_ADDR = a
MR[0] == 0x11
LEQ
DATA <= 0
else
pc = c
Write DATA
pc += 3 + u
Write DATA
MEM_ADDR = b
MR[0] != 0xFF
MR[0] != 0xCC
To CYCLE
else
(subleq, addleq)
MR[1] = 0000 CMP OVERFLOW
Write MR[1]
MEM_ADDR = 0x01
To CYCLE
To STOP
mem[b] == 0x01, HALT
To INT
mem[b] == 0x03
else
To CYCLE
Save MR[mem[b]]
0 < mem[b] < 8
else
To CYCLE
pc += 3 + u
Write DATA
MEM_ADDR = mem[b]
MR[0] == 0x22
else
To LEQ
Set HALT
STOP

Código em Assembly

- Utilização da sintaxe case sensitive, compatível com Linux
- Instanciar os 8 primeiros registradores de máquina
- Instanciação do “program memory”
- Por fim, (opcional) “data memory”

```
# Machine registers section
MCR:    255
CHR:    0
IWR:    0
ICR:    0
CSR:    192
ISR:    0
IDR:    0
IOR:    0
# Program memory section
        exec _SUBLMCR, MCR
        exec _NULL, _NULL -> main
memcpy: exec _SUBLMCR, MCR
        exec _NULL, _NULL
        exec _MEMRMCR, MCR
        exec Var_2_memcpy, m_ptr
        exec _SUBLMCR, MCR
        exec const.0, m_pt
        ...
        exec _MEMMCR, MCR
        exec const.20, Var_33_main
        exec _SUBLMCR, MCR
        exec _NULL, _NULL -> Label_19_main
# Data memory section
_SUBLMCR:         255
_MEMMCR:          34
_MEMRMCR:         17
_NULL:            0
m_ptr:            31500
_NULL:            0
link_register:    0
_TMP:             0
const.0:          1
Var_2_memcpy:     0
Var_33_main:      0
const.1:          2
const.20:         -3
...
```

Modo OISC

- **Exec:** A Single Instruction de 6 bytes
- **MCR:** Definido inicialmente no valor de 255(execução de subleq)
- **Assembly:** Definir todos os 8 registradores no começo do código

```
# Machine registers section
MCR:    255
CHR:    0
IWR:    0
ICR:    0
CSR:    192
ISR:    0
IDR:    0
IOR:    0
# Program memory section
        exec _SUBLMCR, MCR
        exec _NULL, _NULL -> main
memcpy: exec _SUBLMCR, MCR
        exec _NULL, _NULL
        exec _MEMRMCR, MCR
        exec Var_2_memcpy, m_ptr
        exec _SUBLMCR, MCR
        exec const.0, m_pt
        ...
        exec _MEMMCR, MCR
        exec const.20, Var_33_main
        exec _SUBLMCR, MCR
        exec _NULL, _NULL -> Label_19_main
# Data memory section
_SUBLMCR:         255
_MEMMCR:          34
_MEMRMCR:         17
_NULL:            0
m_ptr:            31500
_NULL:            0
link_register:    0
_TMP:             0
const.0:          1
Var_2_memcpy:     0
Var_33_main:      0
const.1:          2
const.20:         -3
...
```

Modo CISC

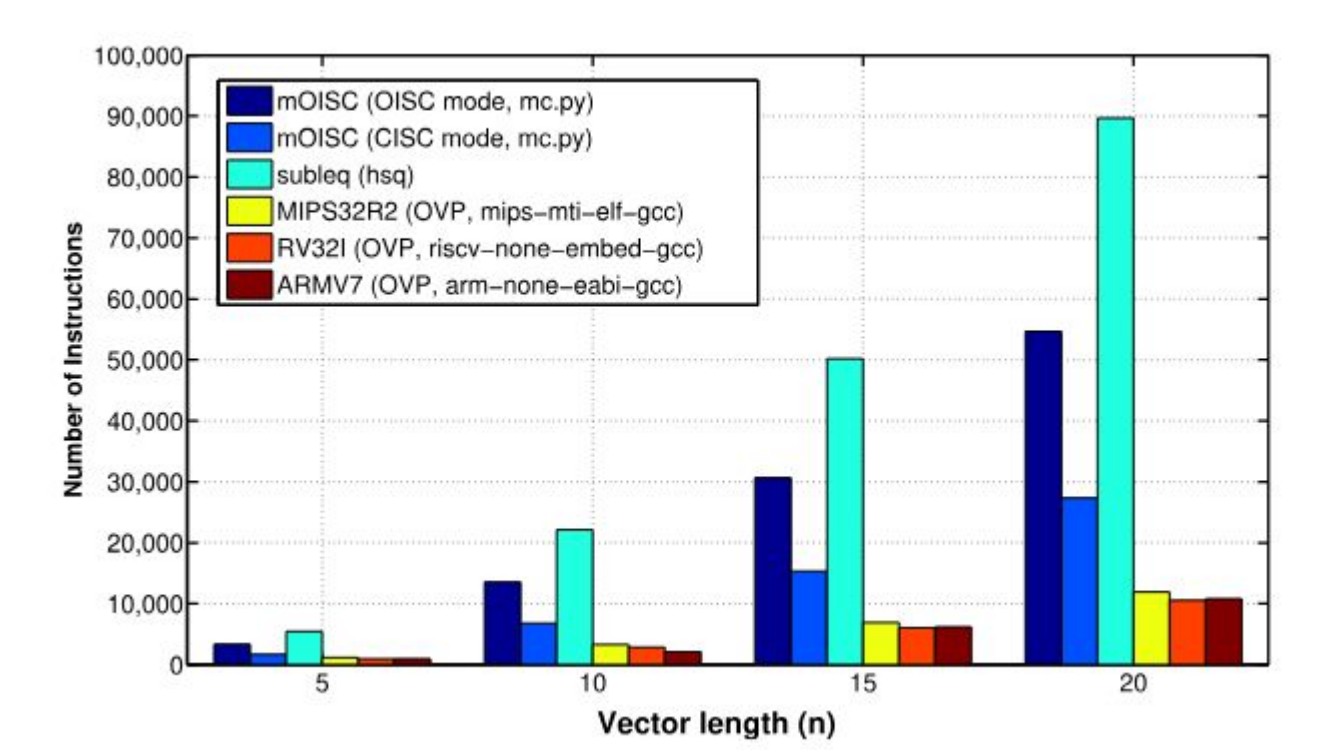
- Instrução recebe quatro endereços MCR,a,b e c
- Definição dinâmica de instrução
- Memory-to-memory

b < 0x08				
MCR	**Mnemonic**	**Name**	**Data Flow**	**Control Flow**
–	MOV	MOVe	`mem[b] = mem[a]`	`pc = c`
b >= 0x08				
MCR	**Mnemonic**	**Name**	**Data Flow**	**Control Flow**
0xFF	SUBLEQ	SUBtract and jump if Less or EQual	`mem[b] = mem[b] - mem[a]`	
0xEE	MOVLEQ	MOVe and jump if Less or EQual	`mem[b] = mem[a]`	
0xCC	ADDLEQ	ADD and jump if Less or EQual	`mem[b] = mem[b] + mem[a]`	
0x99	SHLLEQ	SHift Left and jump if Less or EQual	`mem[b] = mem[a] « mem[b]`	`if mem[b] <= 0:`
0x88	SHRLEQ	SHift Right and jump if Less or EQual	`mem[b] = mem[a] » mem[b]`	`  pc = c`
0x77	ORLEQ	bitwise OR and jump if Less or EQual	`mem[b] = mem[b] | mem[a]`	`else:`
0x66	ANDLEQ	bitwise AND and jump if Less or EQual	`mem[b] = mem[b] & mem[a]`	`  pc += 3 + u`
0x55	XORLEQ	bitwise XOR and jump if Less or EQual	`mem[b] = mem[a] ^ mem[b]`	
0x44	XNORLEQ	bitwise XNOR and jump if Less or EQual	`mem[b] = ~(mem[a] ^ mem[b])`	
0x33	PC	Program Counter save	`mem[b] = pc`	
0x22	MEM	MEMory double-depth addressing	`mem[mem[b]] = mem[a]`	`pc += 3 + u`
0x11	MEMR	MEMory Reverse double-depth addressing	`mem[a] = mem[mem[b]]`	`pc += 3 + u`

0x00	PCS	Program Counter Set	–	`if mem[b] == 0:`
				`  pc = c`
				`else:`
				`  pc = mem[b]`

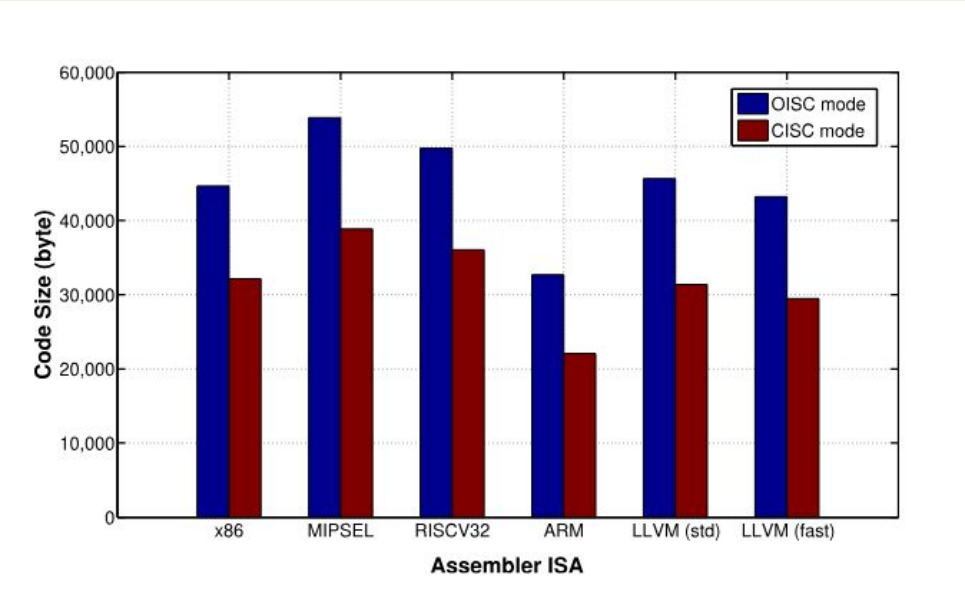
Tabela de decodificação MCR, modo CISC

Análise computacional dos modos OISC x CISC

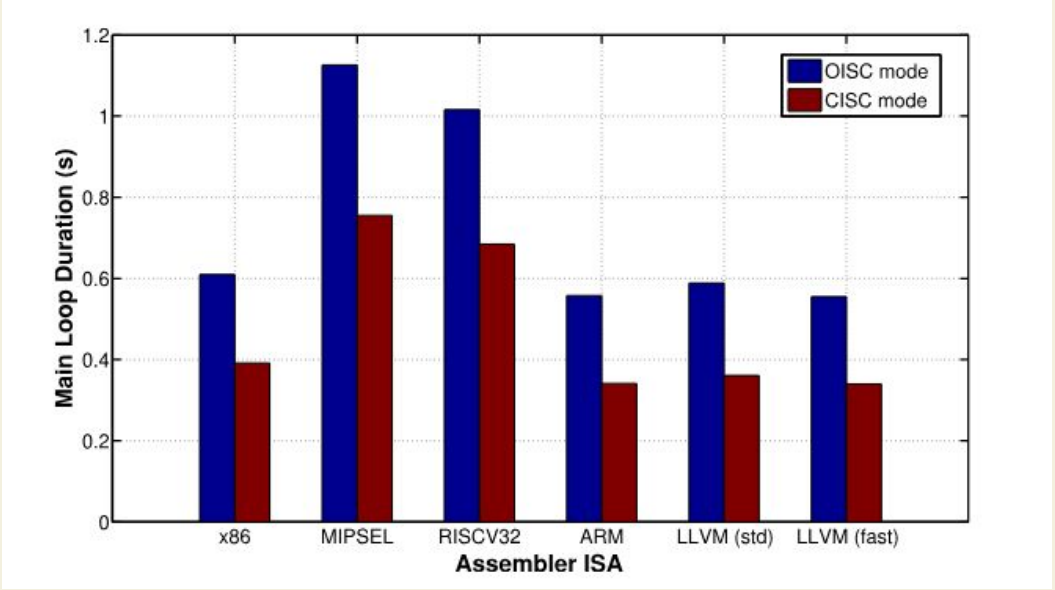


- Parâmetro analisado: análise de complexidade

Desempenho de embarcados dos modos OISC x CISC



- Parâmetro analisado: tamanho código de máquina



- Parâmetro analisado: tempo de execução

Dito isso, qual deles é mais “eficaz”?

Dito isso, qual deles é mais “eficaz”?

depende....

Por que o modo OISC “vence”?

- Hardware mais simples possível
- Eficiência em loops monótonos
- Baixa latência de Fetch de instruções

Por que o modo CISC “vence”?

- Economia de memória, fator ⅔
- Instruções de setup são realizadas em menos ciclos de clock
- Maior compatibilidade com compiladores

Hora da prática!

Vamos aprender a fazer um Hello World em SUBLEQ?

Perguntas?

Referências

Crepaldi, Marco, Andrea Merello, and Mirco Di Salvo. "A multi-one instruction set computer for microcontroller applications." IEEE Access 9 (2021): 113454-113474.

Disponível em: https://ieeexplore.ieee.org/abstract/document/9511494/

Jones, Douglas W. "The ultimate RISC." ACM SIGARCH Computer Architecture News 16.3 (1988): 48-55.

Fonte: https://dl.acm.org/doi/abs/10.1145/48675.48683

Ghosh, Subir, et al. "A complementary two-dimensional material-based one instruction set computer." Nature 642.8067 (2025): 327-335.

Disponível em: https://www.nature.com/articles/s41586-025-08963-7

---

link

Source: www.hackster.io

One Instruction Set Computer, Drexel University

https://www.cs.drexel.edu/~bls96/oisc/

Referências

Mavaddat, Farhad, and Behrooz Parhami. "URISC: the ultimate reduced instruction set computer." International Journal of Electrical Engineering Education 25.4 (1988): 327-334.

Fonte: https://journals.sagepub.com/doi/abs/10.1177/002072098802500408

Whomtech. “Transport Triggered Architecture”

Disponível em : https://whomtech.com/tutorials/advanced-devices/transport-triggered-architecture/

Corporaal, Henk. "Design of transport triggered architectures." Proceedings of 4th Great Lakes Symposium on VLSI. IEEE, 1994.

Disponível em: https://www.cecs.uci.edu/~papers/compendium94-03/papers/1994/glsvlsi94/pdffiles/glsvlsi94_130.pdf

---

link

Source: www.hackster.io

One Instruction Set Computer, Drexel University

https://www.cs.drexel.edu/~bls96/oisc/