27/julho/2025

PUCRS | ESCOLA POLITÉCNICA

4646B-04

FUNDAMENTOS DE SISTEMAS DIGITAIS

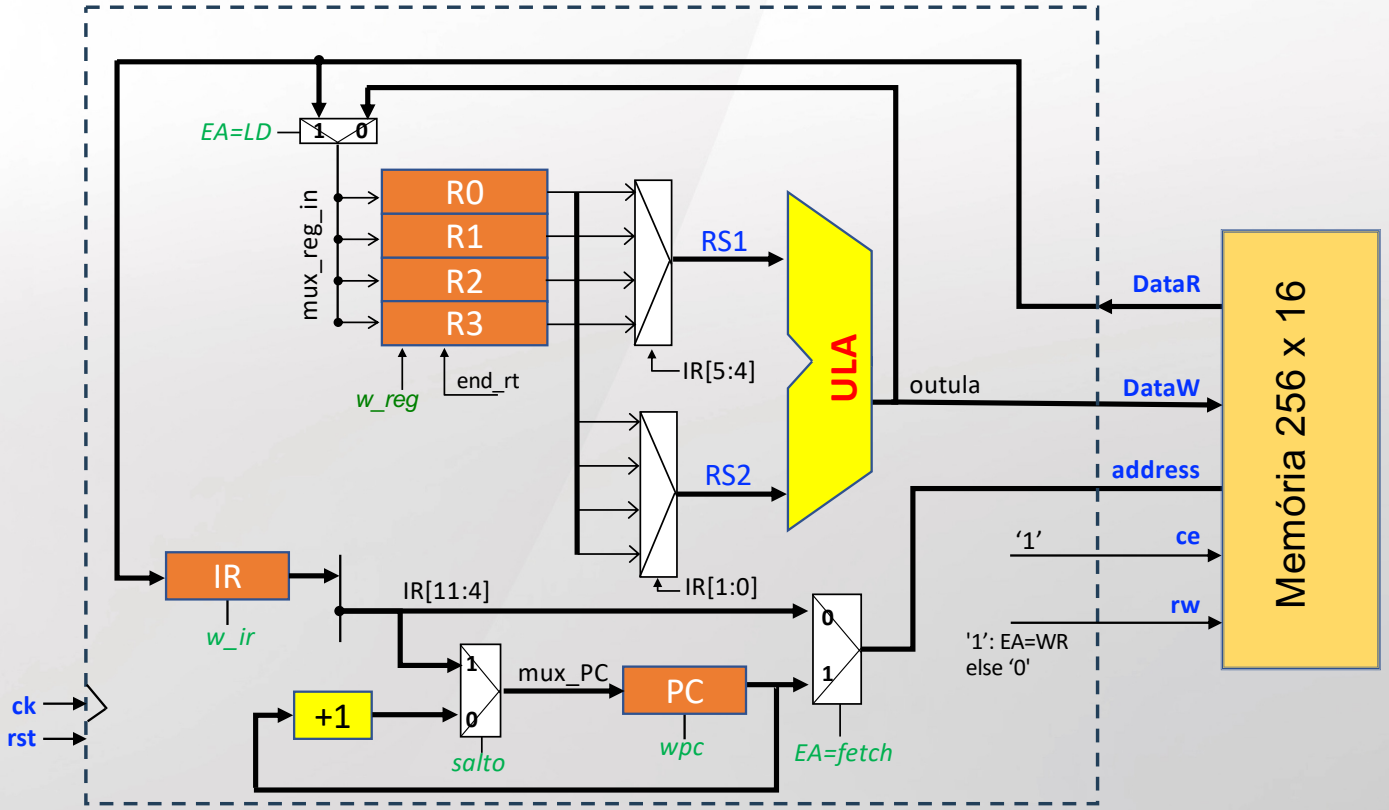
UNIDADE 4
Projeto de Final de Disciplina
NanoCPU (modelagem SV)

MARISTA

Nano CPU

Exemplo de processador muito simples, porém seguindo conceitos importantes em arquitetura de processadores:

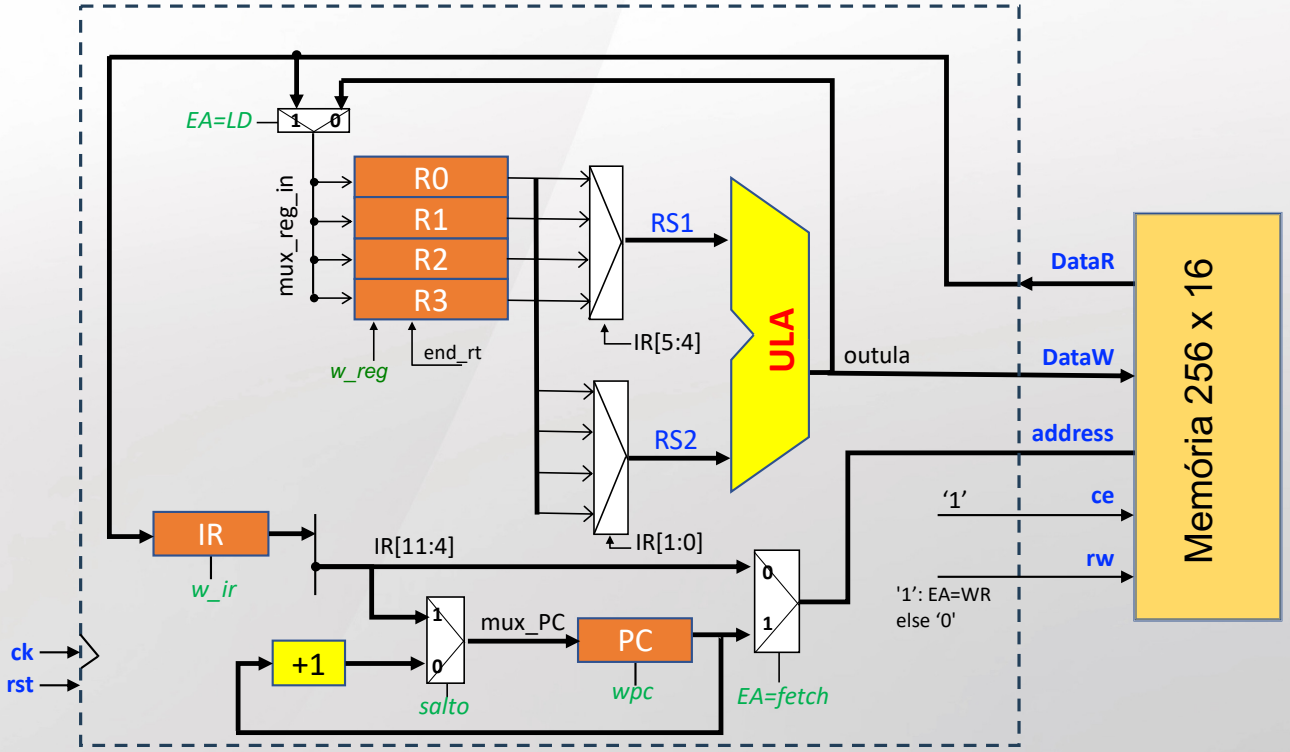
1. Operações lógicas e aritméticas executadas entre registradores
   Arquitetura ***load-store***
2. Instruções de formato fixo



- **R0 a R3 → registradores de propósito geral**
- **PC: *program counter* – endereço atual do programa na memória**
- **IR: *instruction register* – instrução atual**

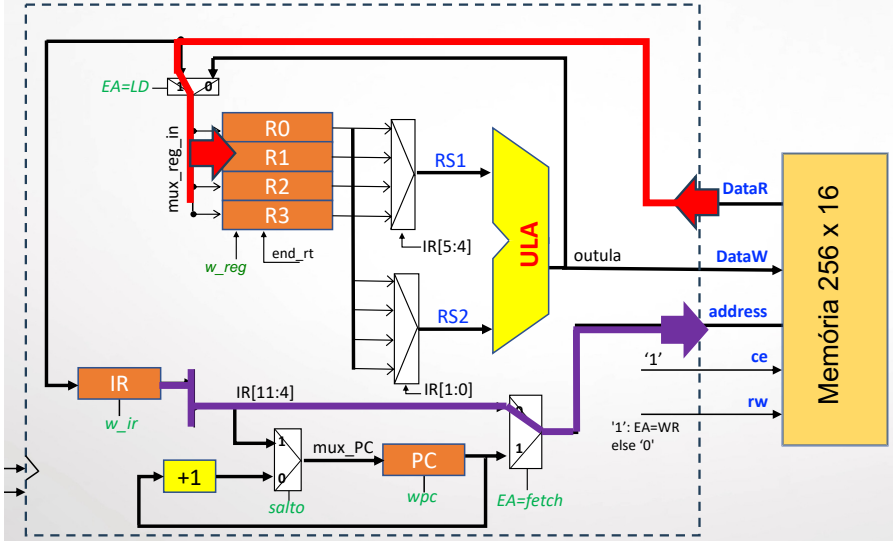
Nano CPU - Interface Externa

```
module NanoCPU (
    input  logic ck, rst,
    output logic [7:0] address,
    input  logic [15:0] dataR,
    output logic [15:0] dataW,
    output logic ce, we
);
```



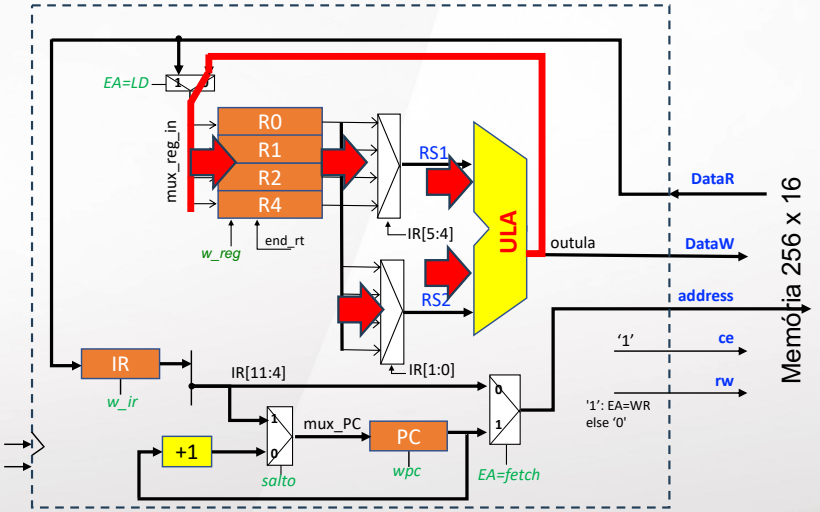
PUCRS ESCOLA POLITÉCNICA

Como processar os dados armazenados em memória?



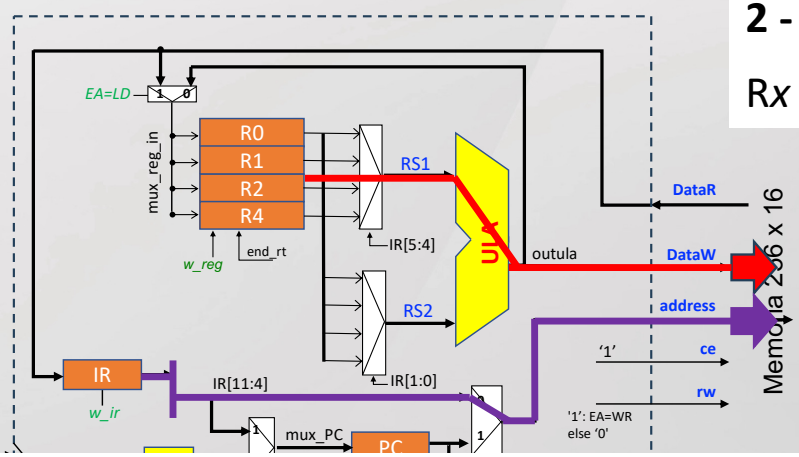
1- Le operandos da memória

- IR contém instrução atual, e nesta instrução há o endereço da memória
- Dado lido é escrito em um registrador



2 - Operações lógico-aritméticas

Rx ← RS1 *op* RS2



3- Escreve o conteúdo de um registrador na memória

Nano CPU – conjunto de instruções

- O registrador IR contém **instruções** de formato **fixo**:
  - Bits 15:12 – 4 bits que especificam a instrução
  - Bits 11:4 – podem indicar um endereço de 8 bits ou endereços de registradores
  - Bits 3:0 – endereço de registrador

Instrução	15:12	11:8	7:4	3:0	Operação
READ	0	endereço		Rs2	Rs2 ← PMEM(end)
WRITE	1	endereço		Rs2	PMEM(end) ← Rs2
SALTO	2	endereço		0	PC ← end
SALTO COND	3	endereço		Rs2	PC ← end **se** Rs2=1
XOR	4	**Rt**	Rs1	Rs2	Rt ← Rs1 xor Rs2
SUB	5	**Rt**	Rs1	Rs2	Rt ← Rs1 - Rs2
ADD	6	**Rt**	Rs1	Rs2	Rt ← Rs1 + Rs2
MENOR	7	**Rt**	Rs1	Rs2	Rt ← 1 **se** Rs1<Rs2 senão 0
...	...				*a ser incluído pelos alunos*
FIM	15 $(F)_{16}$	0	0	0	termina a execução

```
typedef enum logic [3:0] {
        iREAD, iWRITE, iJMP, iBRANCH,
        iXOR, iSUB, iADD, iLESS, iEND
    } instType;
instType inst;
```

Notar que podemos ter até 16 registradores de propósito geral – por que?

Programando o processador

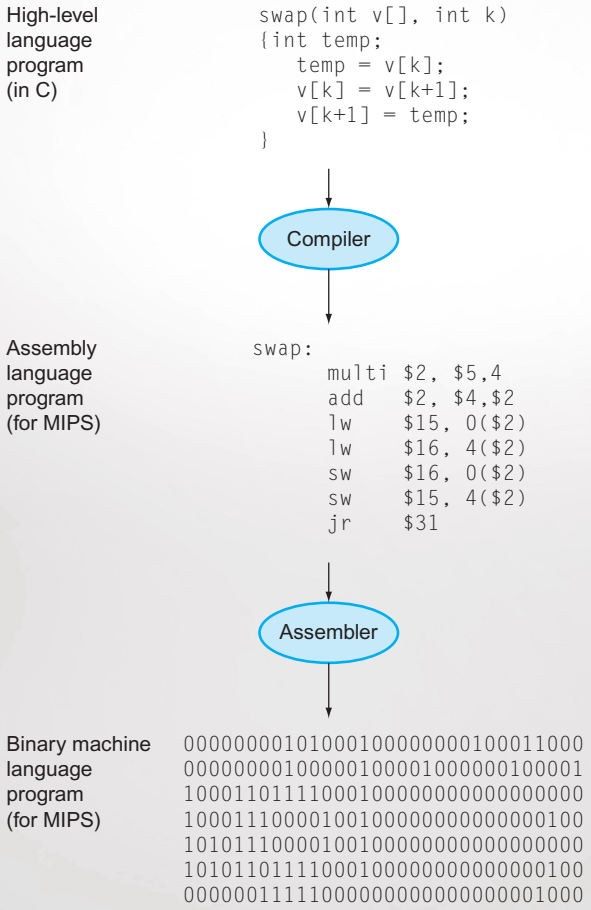


FIGURE 1.4 C program compiled into assembly language and then assembled into binary machine language. Although the translation from high-level language to binary machine language is shown in two steps, some compilers cut out the middleman and produce binary machine language directly. These languages and this program are examined in more detail in Chapter 2.

- A memória contém o programa a ser executado, assim como os dados utilizados no processamento
- Programar direto em binário é impraticável – por isto os processadores usam um linguagem chamada ***assembly***, única para cada processador

Linguagem ***assembly***: descrição de um programa utilizando instruções suportadas pela arquitetura (**ISA** – *Instruction Set Architecture)*

Código binário que vai para a memória do processador

PUCRS ESCOLA POLITÉCNICA

Exemplo de programa (sintaxe informal)

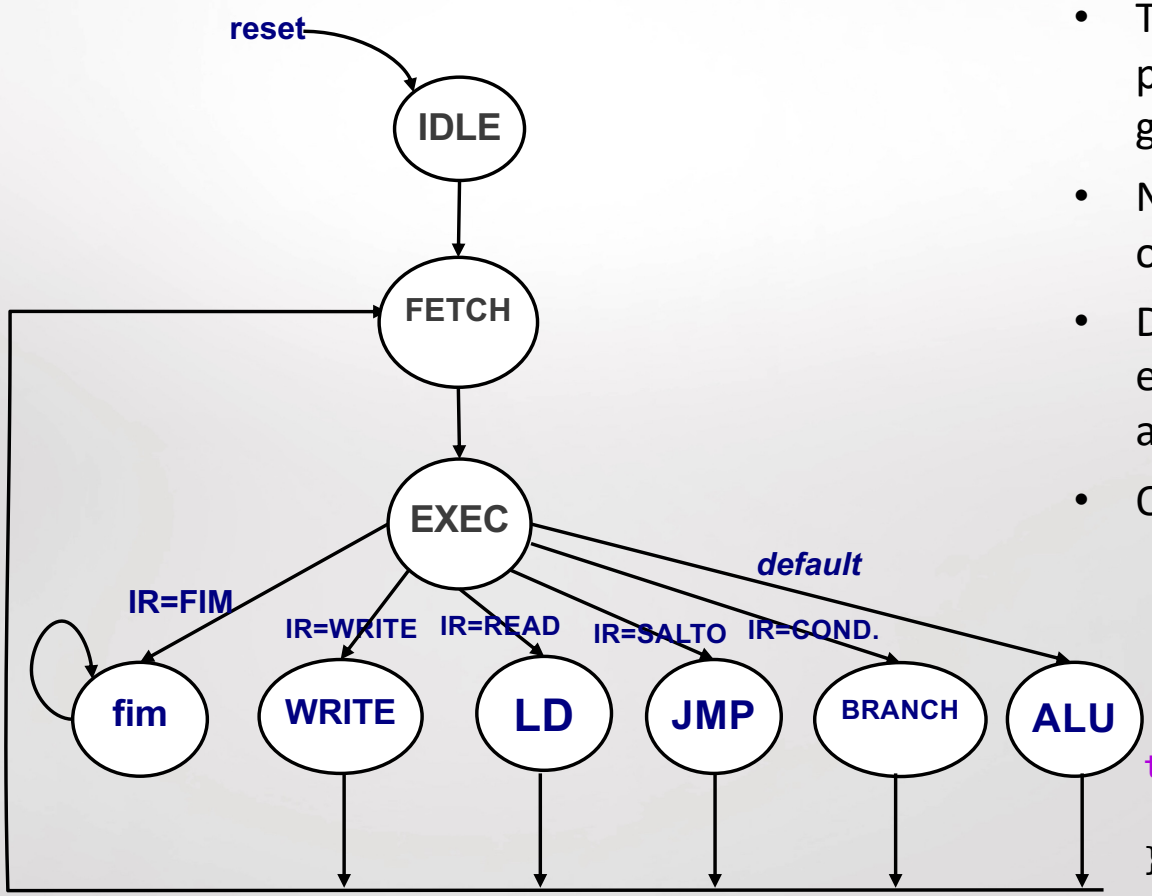
```
i = 0
a =0
do
        a = a +  i
        i  = i + 1
while (i<10)
“write” a
fim
```

Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2
WRITE	1	endereço		Rs2
SALTO	2	endereço		0
SALTO COND	3	endereço		Rs2
XOR	4	**Rt**	Rs1	Rs2
SUB	5	**Rt**	Rs1	Rs2
ADD	6	**Rt**	Rs1	Rs2
MENOR	7	**Rt**	Rs1	Rs2
...	...			
FIM	F (15)	0	0	0

Endereço	Instrução - *assembly*	comentário	**Binário (hexa)**
0	**read** R0, 10	Le da posição 10: R0←0 (i)	0 0A 0
1	**read** R1, 10	Le da posição 10: R1←0 (a)	0 0A 1
2	**read** R3, 12	Le da posição 12: R3 ← 10	0 0C 3
3	**add** R1, R1, R0	a = a + i	6 1 1 0
4	**read** R2, 11	Le da posição 11: R2 ← 1	0 0B 2
5	**add** R0, R0, R2	i = i + 1	6 0 0 2
6	**menor** R2, R0, R3	R2←1 se i<10 else 0	7 2 0 3
7	**salto cond** 3, R2	se R2=1 salta para 3	3 0 3 2
8	**write** R1, 13	Escreve a na posição 13	1 0D 1
9	**fim**	Termina a execução	F000
10 $(A)_{16}$	0		0000
11 $(B)_{16}$	1		0001
12 $(C)_{16}$	10		000A
13 $(D)_{16}$		Receberá o valor de a	0000

Como o processador executa um programa?

Máquina de estados para sequenciamento das instruções e sinais de controle



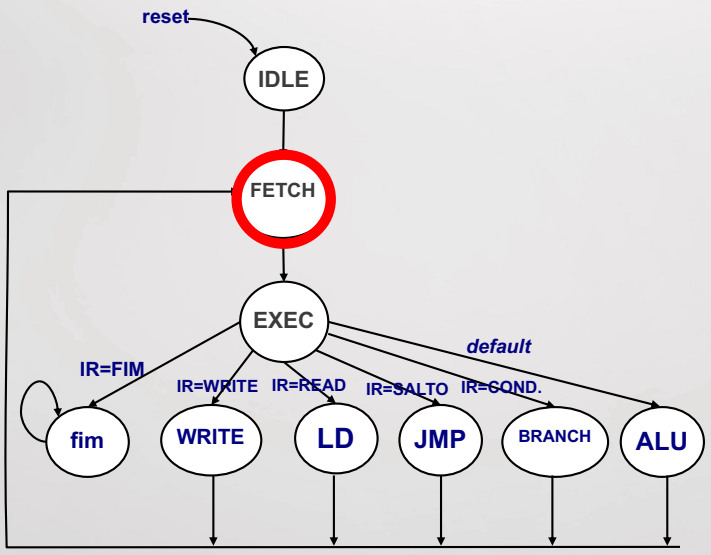
- Todas as instruções executam o estado de ***fetch*** (busca) para ler da memória a instrução apontada pelo **PC** e gravar este dado no registrador **IR**
- No estado ***exec*** é feita a leitura da memória ou a operação na ULA
- Depois cada instrução é concluída em um estado específico (por default as instrução lógicas e aritméticas são executadas no estado ***ALU***)
- O programa termina ao encontrar a instrução **fim**

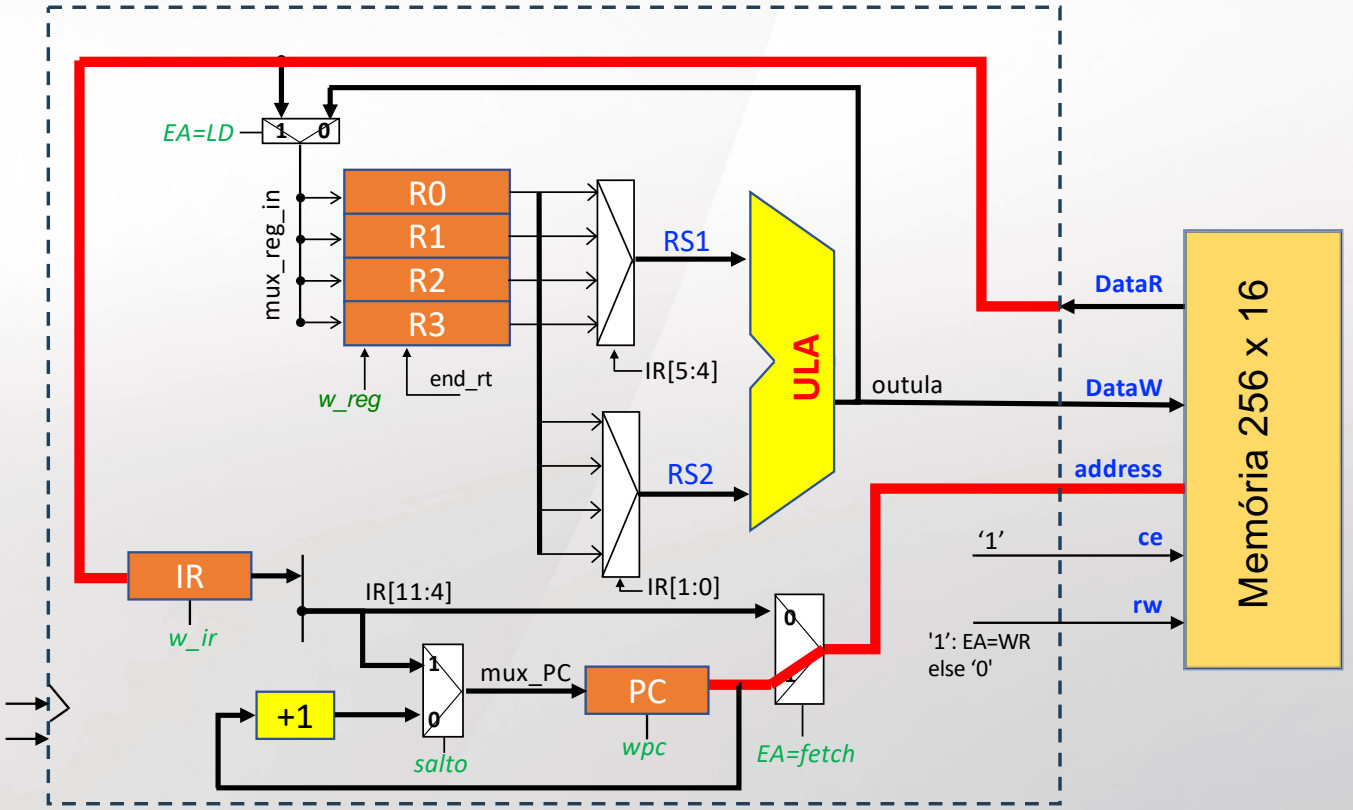
```
typedef enum logic [3:0] {
   IDLE, FETCH, EXEC, LD, WRITE, ALU, JMP, BRANCH, fim
} EAType;
EAType EA;       // apenas EA (código + simples)
```

Busca da instrução – igual para todas as instruções

Neste estado de ***fetch***:

- PC endereça a memória
- Dado é lido da memória
- Ativa o *w_ir* para escrita em IR





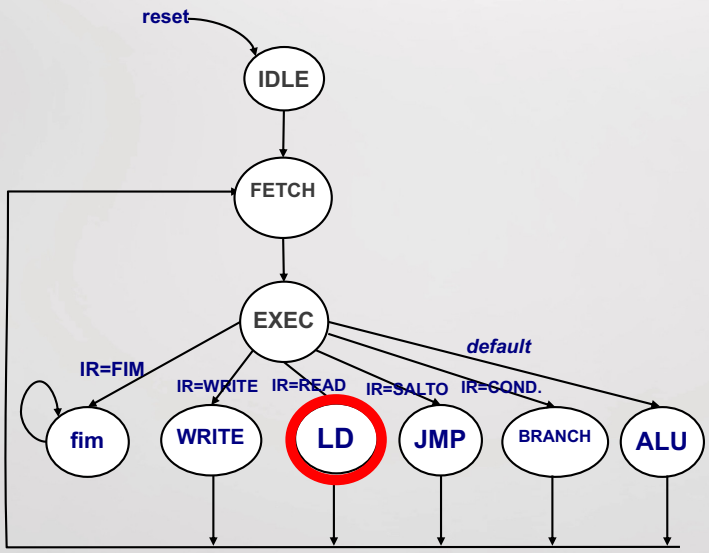
Busca da instrução - *fetch*

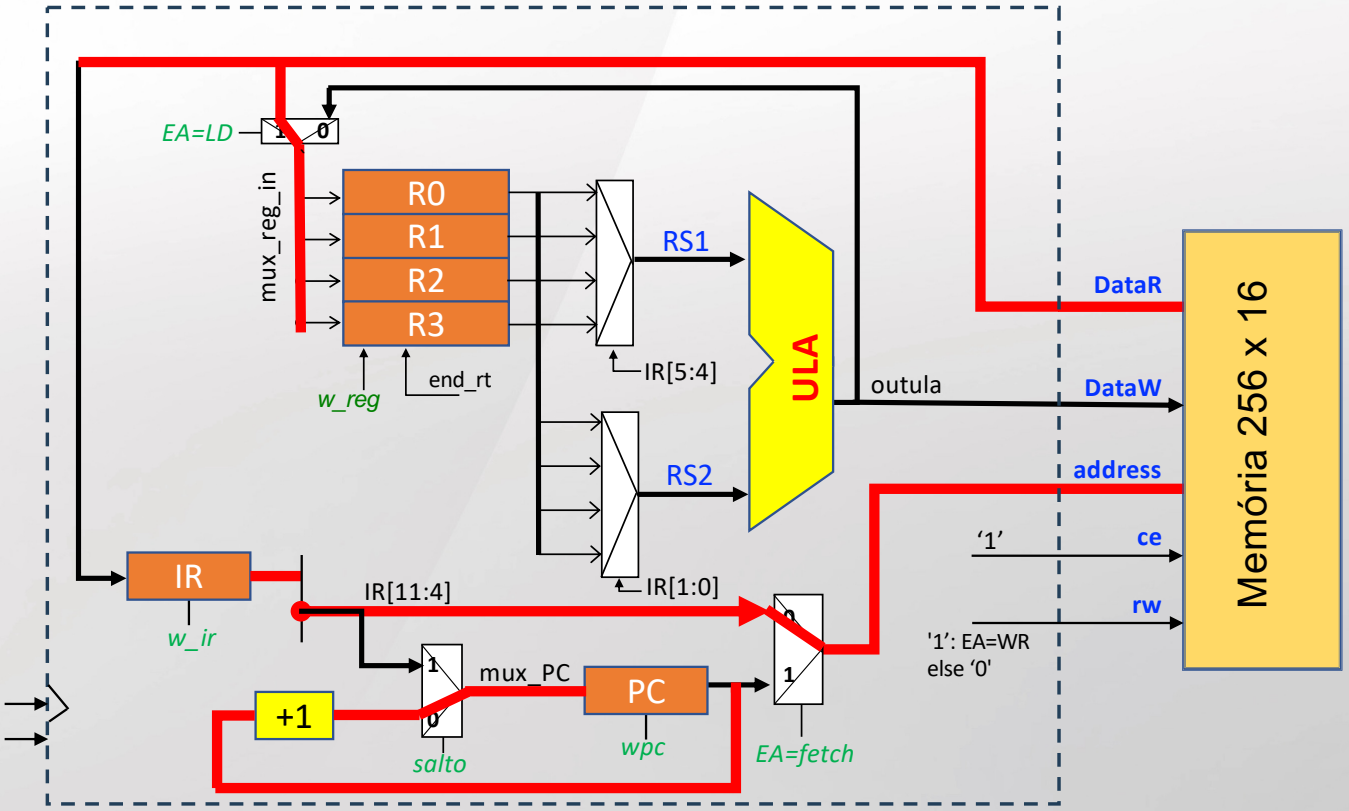
Execução do READ

No estado ***LD***:

- IR[11:4] endereça a memória
- Dado é lido da memória
- Escreve no banco de registradores ativando ***w_reg*** (registrador alterado especificado em IR[1:0])
- Incrementa o PC (ativa ***wpc***)

Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2





Simulação

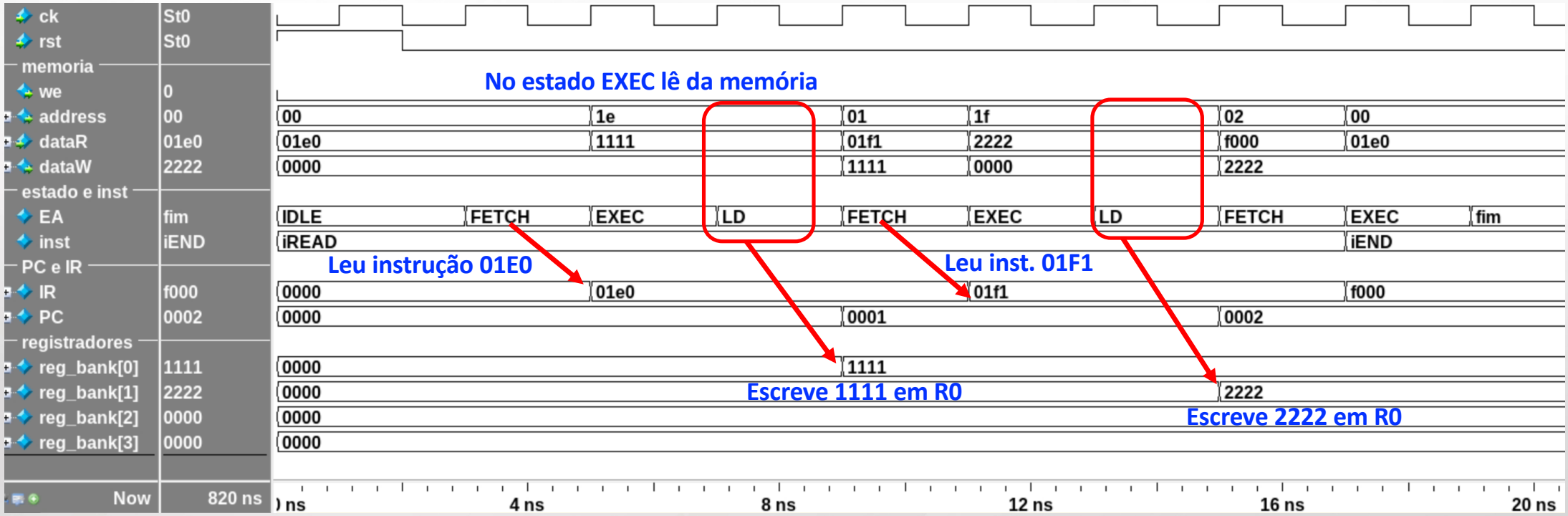
- Baixar do Moodle o arquivo nano_cpu.zip

```
.
├── nanoCPU.sv
├── nanoTB.sv
├── sim.do
└── wave.do
```

- Executar o sim.do fornecido, vizualizando as formas de onda entre 0 e 20 ns

PUCRS ESCOLA POLITÉCNICA

Inicializando dois registradores




```
memory_array_t memory = '{
  0:  'h01E0, // R0 = PMEM[30]
  1:  'h01F1, // R1 = PMEM[31]
  2:  'hF000, // FIM
  30: 'h1111,
  31: 'h2222,
  default: 'h0000
};
```

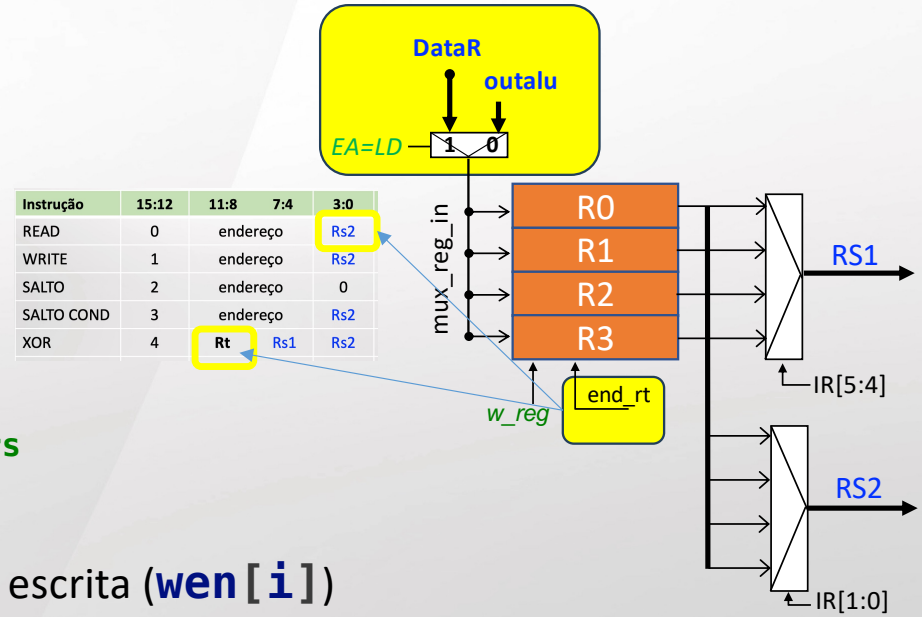
O memory_array armazena o binário (programa)
a estrutura é <endereço decima> : <valor>

Atividade 1 completar o banco de registradores (a)

```
// register bank - 4 general purpose registers
genvar i;
generate
   for (i = 0; i < 4; i++) begin
      Reg16bit reg_inst (.ck(ck), .rst(rst), .we(wen[i]),
                         .D(muxRegIn),.Q(reg_bank[i]) );
      assign wen[i] = (addReg==i && wReg) ? 1 : 0;
   end
endgenerate

assign addReg = IR[1:0];    // mux ausente
assign muxRegIn = dataR;    // mux ausente

assign RS1 = reg_bank[IR[5:4]]; // reg bank output multiplexers
assign RS2 = reg_bank[IR[1:0]];
```



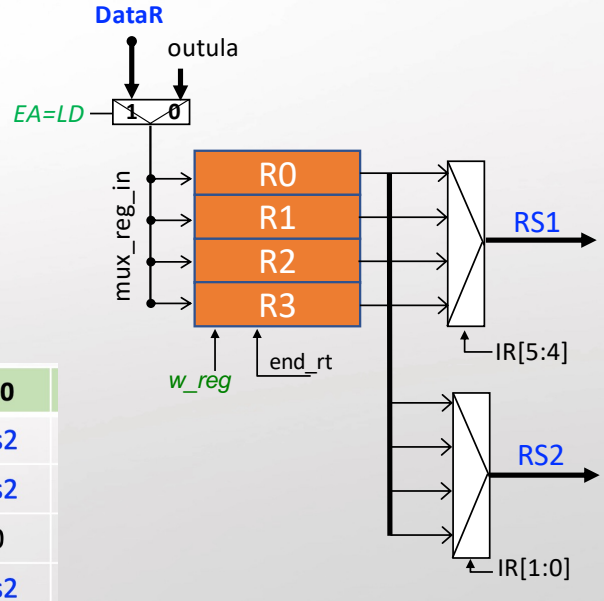
- **generate**: instancia os 4 registradores e defina o sinal de escrita (**wen[i]**)
- Completar os dois multiplexadores ausentes na descrição fornecida
  - **addReg** : deve receber IR[1:0] apenas no estado LD, por default deve receber IR[9:8]
  - **muxRegIn** : deve receber o dado da memória (**dataR**) apenas no estado LD, por default deve receber **outalu**

Atividade 1: completar o banco de registradores (b)

Alterado o banco de registradores, executar o programa para inicializar os 4 registradores (alterando o *test bench*):

```
memory_array_t memory = '{
    0:  'h01E0,      // R0 = PMEM[30]
    1:  'h01F1,      // R1 = PMEM[31]
    2:  completar,   // R2 = PMEM[32]
    3:  completar,   // R3 = PMEM[34]
    4:  'hF000,      // FIM
    30: 'h1111,
    31: 'h2222,
    32: 'h3333,
    33: 'h4444,
    default: 'h0000
};
```

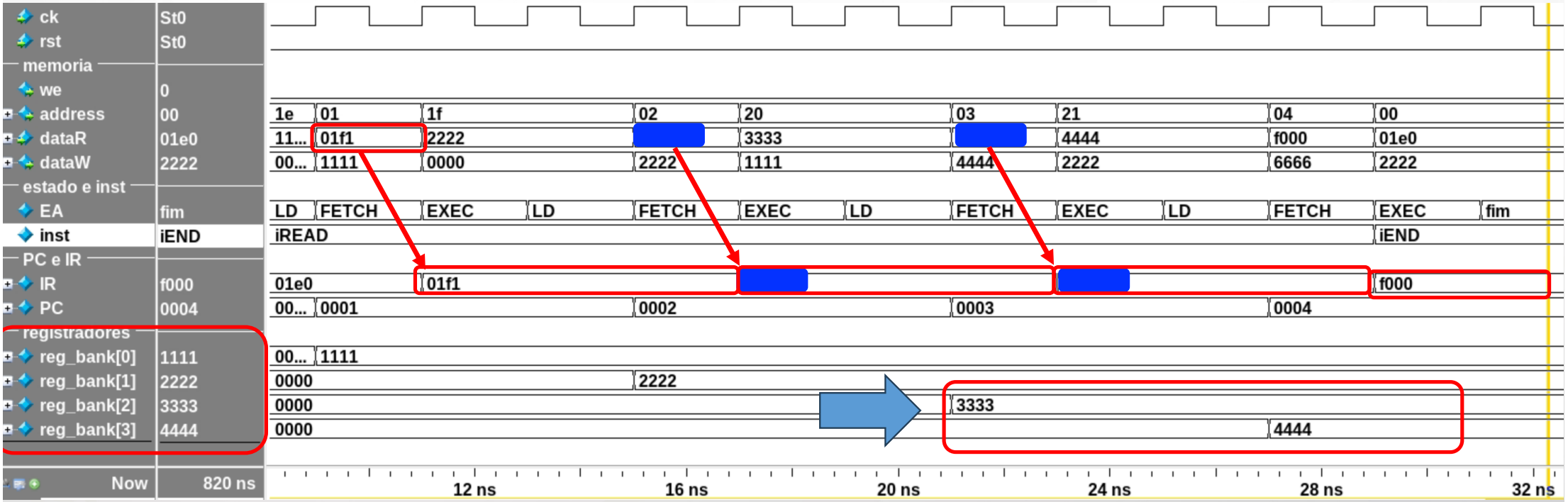
Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2
WRITE	1	endereço		Rs2
SALTO	2	endereço		0
SALTO COND	3	endereço		Rs2
XOR	4	**Rt**	Rs1	Rs2
SUB	5	**Rt**	Rs1	Rs2
ADD	6	**Rt**	Rs1	Rs2
MENOR	7	**Rt**	Rs1	Rs2
...	...			
FIM	F (15)	0	0	0



PUCRS ESCOLA POLITÉCNICA

Atividade 1: completar o banco de registradores (c)

- Simular de forma a inicializar corretamente o **reg[2] e reg[3]** – zoom entre 0 e 35 ns



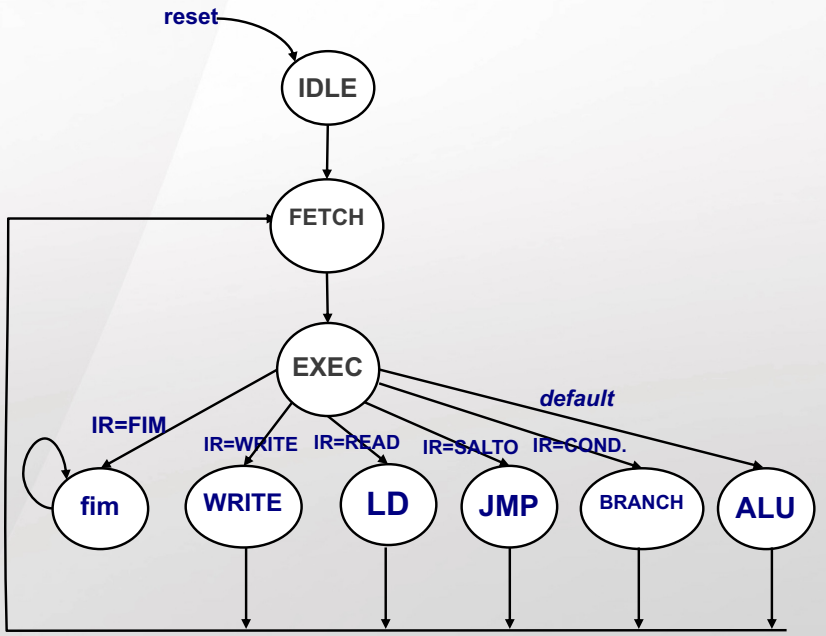
Atividade 2 decodificar instruções

- **Decodificar** as instruções da nanoCPU. O código fornecido decodifica apenas as instruções `iREAD` e `iEND`

```
always_comb begin // decode the current instruction
    case (IR[15:12])
        'h0: inst = iREAD;
        'h1: inst = completar;
        'h2: inst = completar;
        'h3: inst = completar;
        'h4: inst = completar;
        'h5: inst = completar;
        'h6: inst = completar;
        'h9: inst = completar;
       default: inst = iEND;
    endcase
end
```

Atividade 3 completar a máquina de estados

```
always_ff @(posedge ck or posedge rst) begin
    if (rst)
        EA <= IDLE;
    else begin
        unique case (EA)
            IDLE:  EA <= FETCH;
            FETCH: EA <= EXEC;
            EXEC: unique case (inst)
                    iEND:    EA <= fim;
                    iREAD:   EA <= LD;
                    iWRITE:  EA <= completar;
                    iJMP:    EA <= completar;
I                   BRANCH:  EA <= completar;
                    default: EA <= ALU;
                endcase
            fim: EA <= fim;
            default: EA <= FETCH;
        endcase
    end
end
```



Atividade 4 completar ULA

- **ULA** : a ULA fornecida só executa por default a soma. Faltam ao menos 4 instruções

```
// arithmetic and logic unit
always_comb begin
    unique case (inst)
        iWRITE: outalu = completar;
        // completar
        default: outalu = RS1 + RS2;
    endcase
end
```

Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2
WRITE	1	endereço		Rs2
SALTO	2	endereço		0
SALTO COND	3	endereço		Rs2
XOR	4	Rt	Rs1	Rs2
SUB	5	Rt	Rs1	Rs2
ADD	6	Rt	Rs1	Rs2
MENOR	7	Rt	Rs1	Rs2
...	...			
FIM	F (15)	0	0	0

Dica: para **iLess (menor)**: outalu = (RS1 < RS2) ? 'h0001 : 'h0000;

Atividade 5 simulação para validar as alterações

- **Programa** de teste

Já temos:

R0 ← 1111

R1 ← 2222

R2 ← 3333

R3 ←4444

Vamos programar (acrescentar o código binário no testbench)

R0 ← R0 + R3 - espera-se 5555 em R0

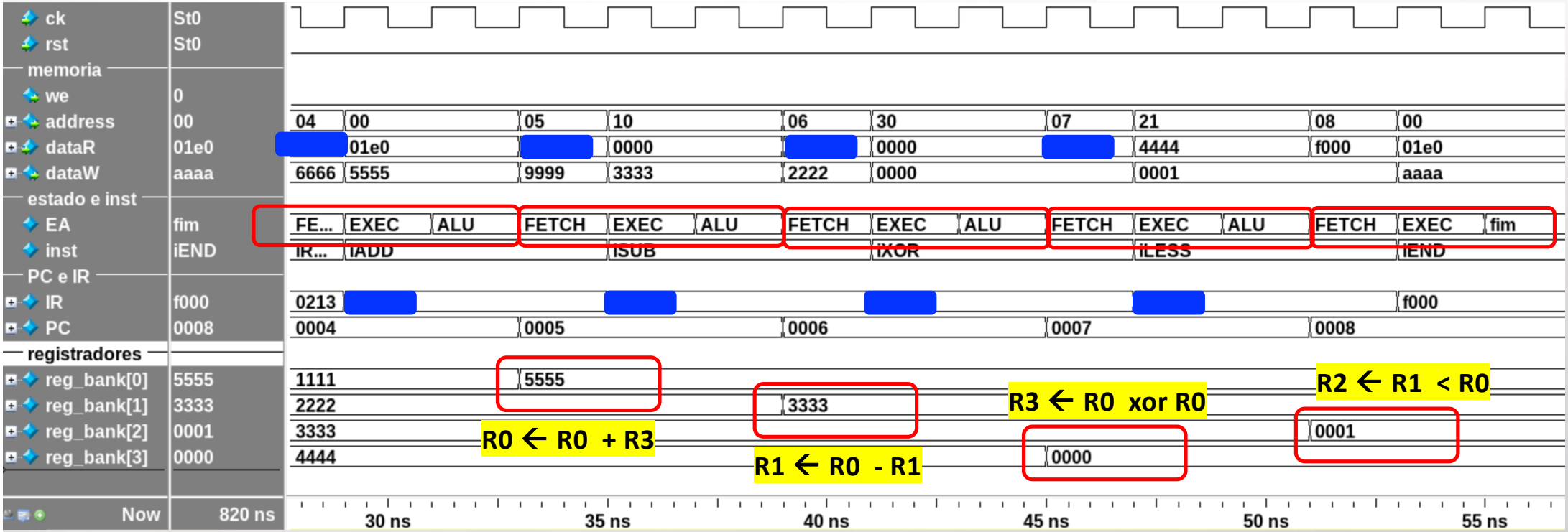
R1 ← R0 – R1 - espera 3333 em R1

R3 ← R0 xor R0 - espera-se que o R3 seja **zerado** (método comum para zerar registradores)

R2 ← R1 < R0 - 3333 < 5555, logo R2 deve ser 1

Atividade 5 simulação para validar as alterações

- Após fazer as alterações no código, e escrever as 4 instruções no *testbench* devemos obter:



Atividade 6 teste de escrita na memória

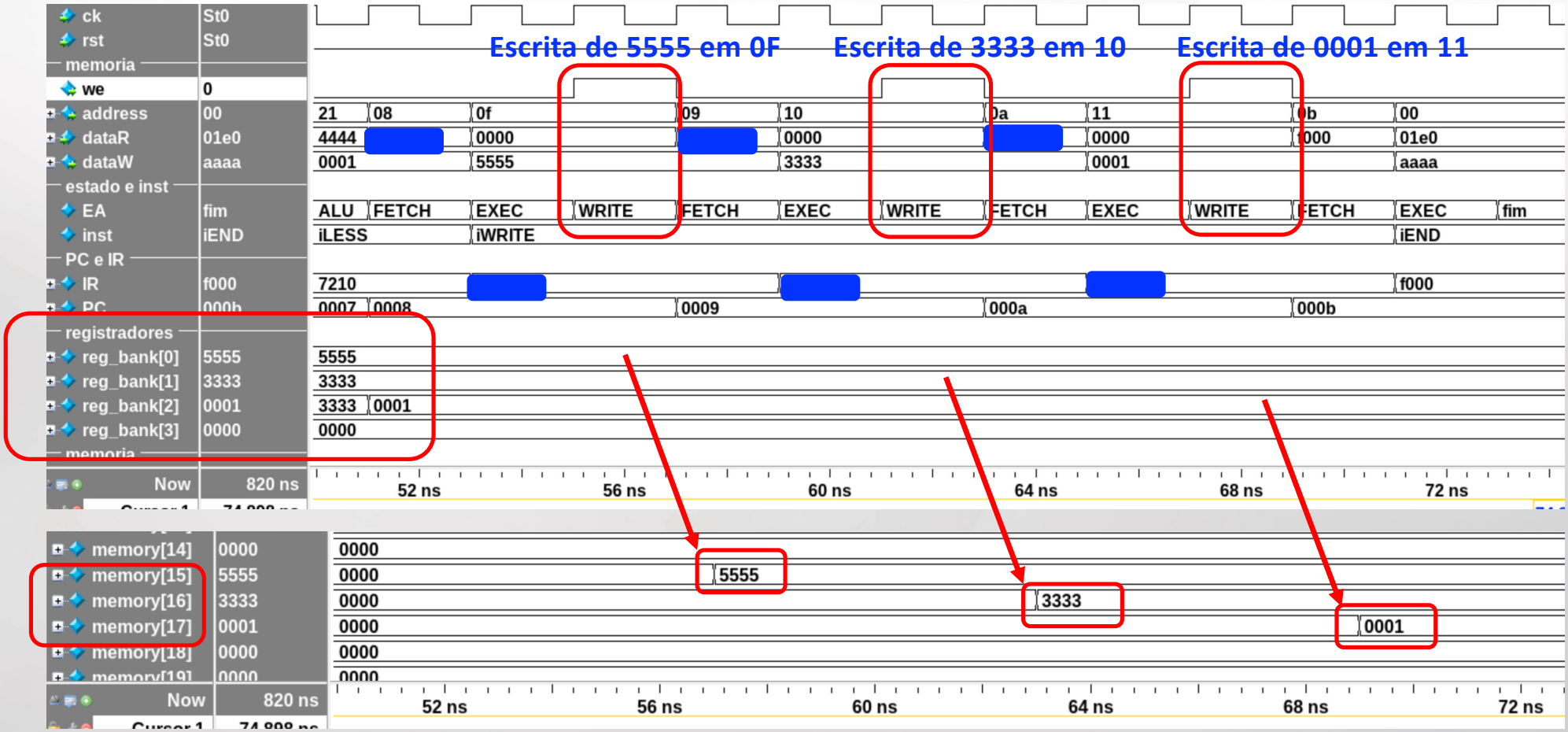
- O sinal **we** está assim definido no código fornecido:

```
assign we = 0; // atividade 6
```

- Este sinal deve ser ativo (1) quando o EA for WRITE, senão 0.

Atividade 6 teste de escrita na memória

Escrever o conteúdo dos registradores R0 (5555), R1 (3333), R2 (0001) na posições 15 $(0F)_{16}$, 16 $(10)_{16}$ e 17 $(11)_{16}$



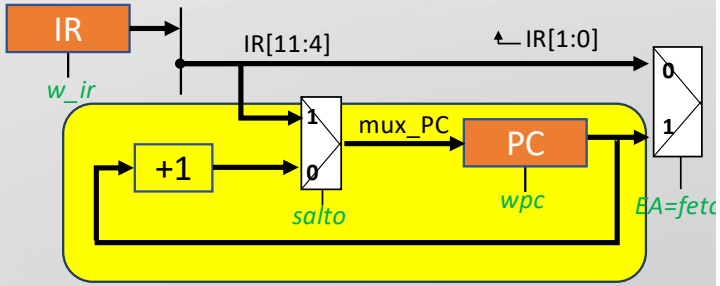
Atividade 7 — Saltos – condicionais e incondicionais

No código fornecido o PC está sendo sempre incrementado:

```
assign muxPC = PC + 1;
```

Deve-se implementar o **muxpC** de tal forma que o PC recebe os 8 bits do IR: `{8'h00, IR[11:4]}`

Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2
WRITE	1	endereço		Rs2
SALTO	2	endereço		0
SALTO COND	3	endereço		Rs2
XOR	4	Rt	Rs1	Rs2



Condições para a atribuição:

(1) EA == JMP

(2) EA == BRANCH && RS2[0] // bit do less ativo

Atividade 7: Saltos – condicionais e incondicionais

Lembrando: o R2 tem $(0001)_{16}$

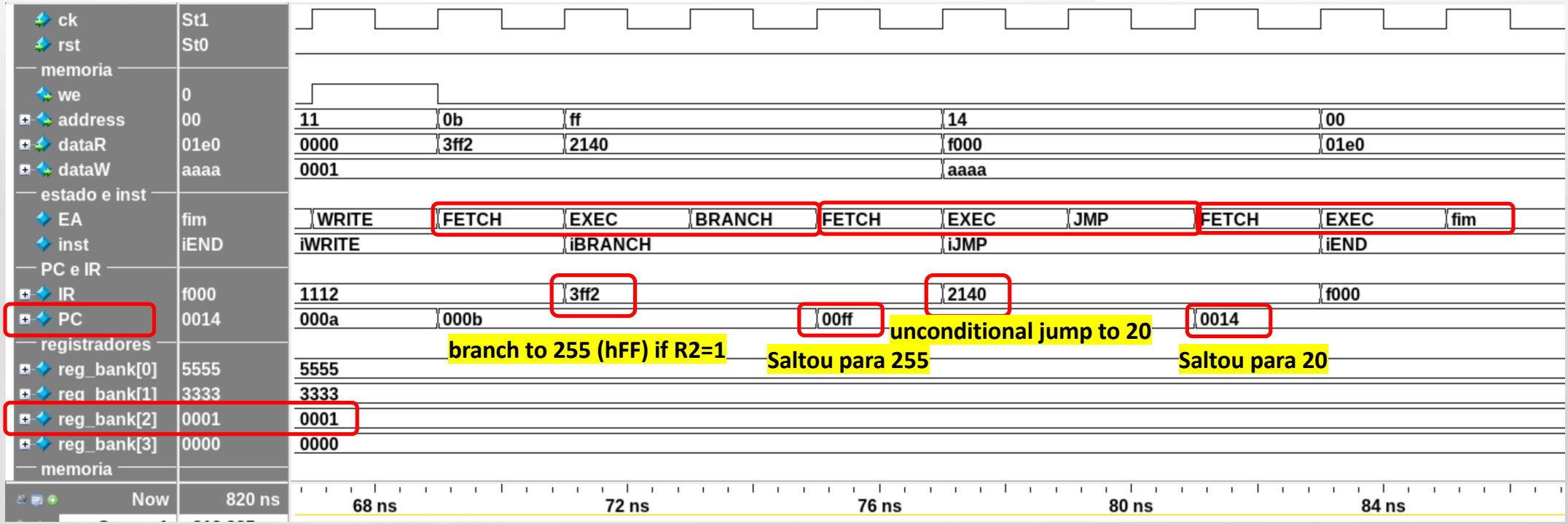
Após o armazenamento de R2:

- salto condicional para endereço 255 se R2=1 **(11=> 'h3FF2)**
- no endereço 255 salta para endereço 20 $(14)_{16}$ **(255=> 'h2140)**
- no endereço 20 termina o programa **(20=> 'hF000)**

Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2
WRITE	1	endereço		Rs2
SALTO	2	endereço		0
SALTO COND	3	endereço		Rs2
XOR	4	**Rt**	Rs1	Rs2

```
   10: 'h1112, // store R2 in 11
   11: 'h3FF2, // branch to 255 (hFF) if R2=1
   20: 'hF000, // end of program
   30: 'h1111,
   31: 'h2222,
   32: 'h3333,
   33: 'h4444,
  255: 'h2140; // unconditional jump to 20 (h14)
default: 'h0000
};
```

Atividade 7: Saltos – condicionais e incondicionais



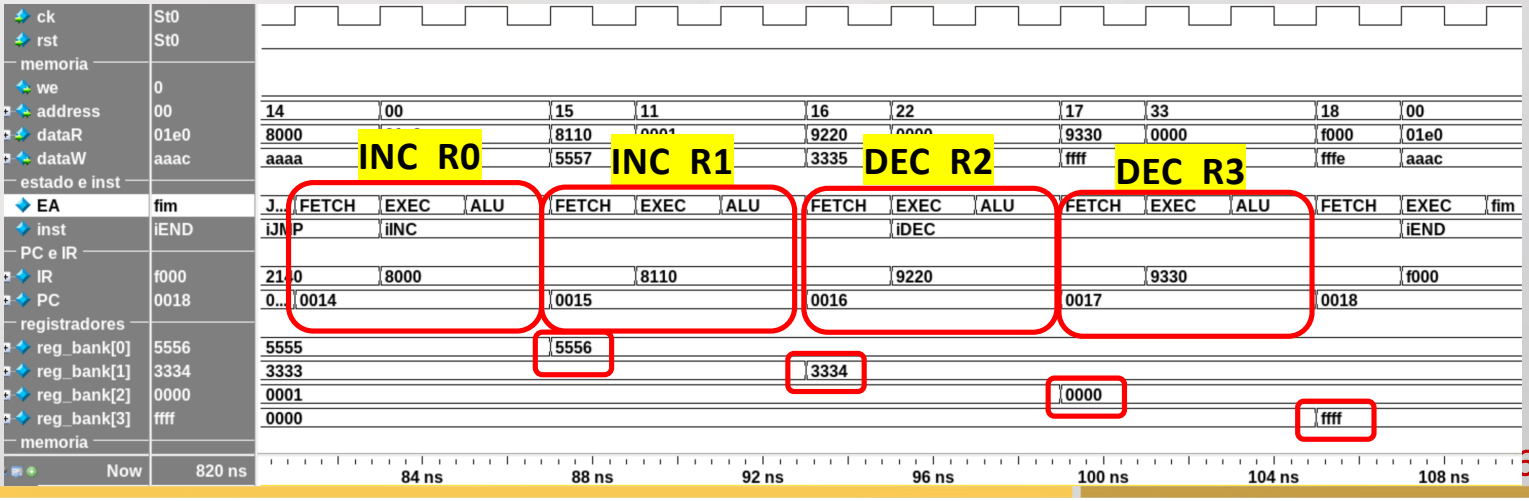
PUCRS ESCOLA POLITÉCNICA

Atividade 8 Incluir instruções INC e DEC

- Incluir duas instruções – **iINC** e **iDEC**
- Usar os campos 8 e 9 para as instruções
  - INC: Rt ← RS1 +1, exemplo **'h8110** R1 ← R1 +1
  - DEC: Rt ← RS1 - 1, exemplo **'h9330** R3 ← R3 - 1
- Ações:
  1. **insere na lista de instruções – `instType` – `iINC, iDEC`**
  2. **acrescentar as operações de incremento e decremento na ULA**
  3. **Decodificar as instruções `iINC` e `iDEC`**
- Programa teste:

```
11: 'h3FF2, // branch to 255 (xFF) if R2=1
20: 'h8000, // INC R0
21: 'h8110, // INC R1
22: 'h9220, // DEC R2
23: 'h9330, // DEC R3
24: 'hF000,
.....
```

Instrução	15:12	11:8	7:4	3:0
READ	0	endereço		Rs2
WRITE	1	endereço		Rs2
SALTO	2	endereço		0
SALTO COND	3	endereço		Rs2
XOR	4	Rt	Rs1	Rs2
SUB	5	Rt	Rs1	Rs2
ADD	6	Rt	Rs1	Rs2
MENOR	7	Rt	Rs1	Rs2
...	...			
FIM	F (15)	0	0	0



Atividade 9 executar o programa exemplo (a)

```
i = 0
a =0
do
      a = a +  i
      i  = i + 1
while (i<10)
“write” a
fim
```

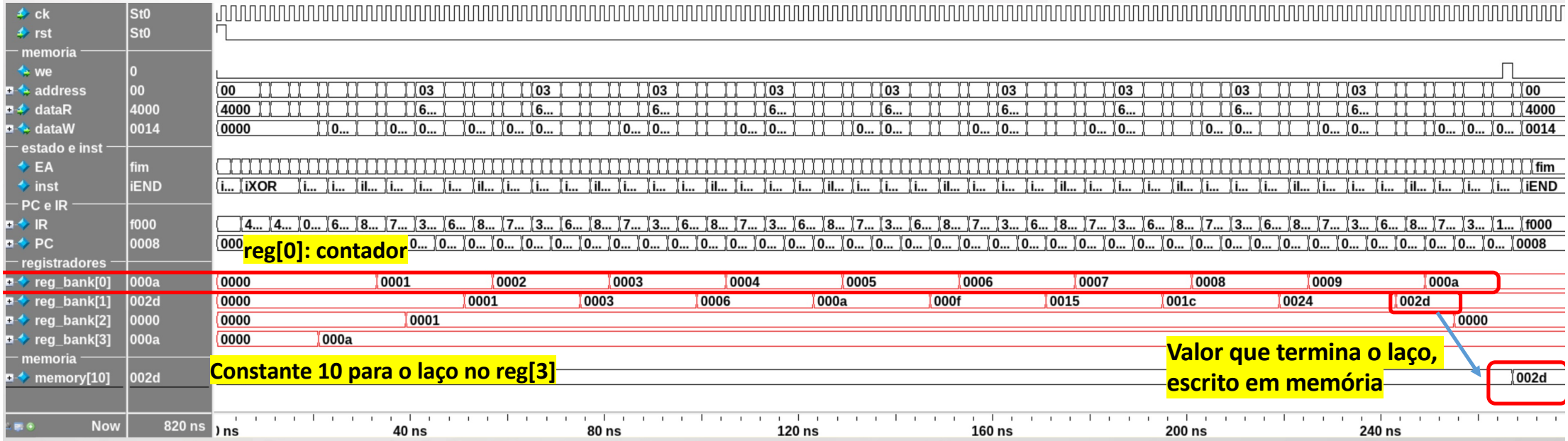
```
memory_array_t memory = '{
    0: 'h4000,
    1: 'h4111,
    2: 'h0093,
    3: 'h6110,
    4: 'h8000,
    5: 'h7203,
    6: 'h3032,
    7: 'h10A1,
    8: 'hF000,
    9: 'h000A,
    default: 'h0000
};
```

Endereço	Instrução - *assembly*	comentário	Binário (hexa)
0	**xor** R0, R0, R0	R0 ←0 (i)	4 0 0 0
1	**xor** R1, R1, R1	R1 ←0 (a)	4 1 1 1
2	**read** R3, 11	Le da posição 09: R3 ← 10	0 09 3
3	**add** R1, R1, R0	a = a + i	6 1 1 0
4	**inc** R0, R0	R0 ← R0 + 1	8 0 0 0
5	**menor** R2, R0, R3	R2←1 se i<10 else 0	7 2 0 3
6	**salto cond** 3, R2	se R2=1 salta para 3	3 0 3 2
7	**write** R1, 10	Escreve a na posição 10	1 0A 1
8	**fim**	Termina a execução	F000
9	10		000A
10 (A)		Receberá o valor de a	

PUCRS ESCOLA POLITÉCNICA

Atividade 9: executar o programa exemplo (b)

- Observar o **reg[0[** – variável i, conta de 0 a 9, no valor 10 termina o laço
- O **reg[1]** é o "**a = a + i**" – 0, 1, 3, 6, 10, 15, 21(x15), 28 (x1C), 36 (x24), 45 (x2D)
- No final grava na posição **10** da memória o valor 0x2D



Tempo de simulação : 300 ns

Atividade 10 Calcular a sequência de Fibonacci (a)

```
fib1 = 0
fib2 = 1

do {
      escreve fib1 na memória
      next = fib1 + fib2
      fib1 = fib2
      fib2 = next
      N = N -1
} while (N>0)
```

Dicas:

- Reservar **R0** como uma constante 0 (zero)

```
'h4000, // R0 <- 0 (constant)
```

- Dois registradores devem ser dedicados a fib1 (**R1**) e fib2 (**R2**)
- Para fazer fib1 = fib2 pode-se fazer um xor com 0

```
'h4120, // R1 <- R2 xor R0 (fib1 <- fib2)
```

- **R3** – usar para tratar *next,* N (lê N da memória, decrementa N, grava N na memória), e usa para o salto condicional (R3=1 se N<0) – é o registrador "temporário"

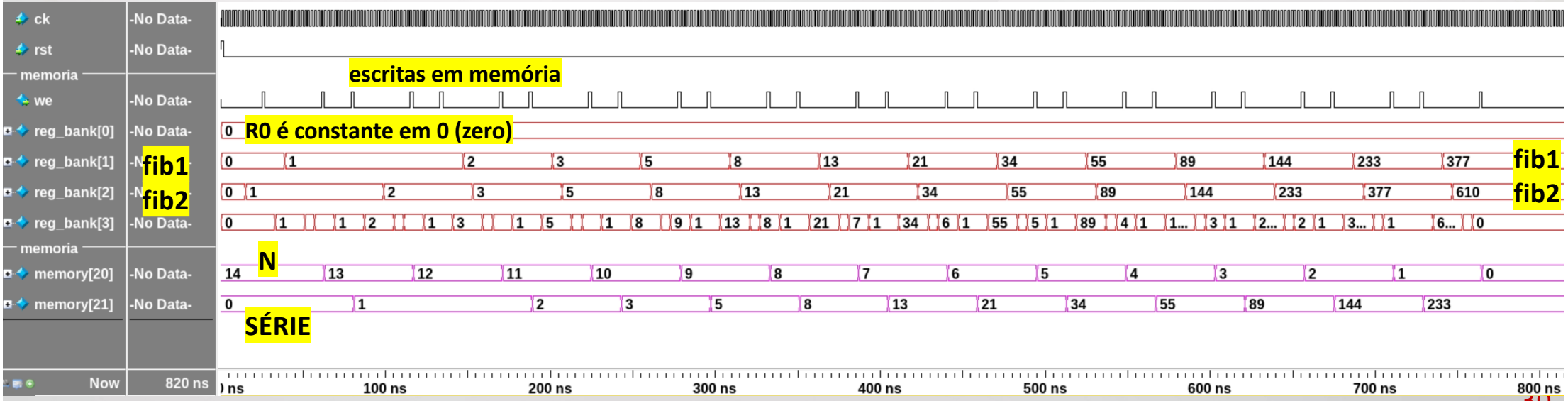
```
memory_array_t memory = '{ // fibonacci
    …
    20: 'h000E, // 14 first elements of the series
    21: 'h0000, // receives the values of the series
    default: 'h0000
};
```

Calcular a sequência de Fibonacci (b)

```
fib1 = 0
fib2 = 1

do {
        escreve fib1 na memória
        next = fib1 + fib2
        fib1 = fib2
        fib2 = next
        N = N -1
} while (N>0)
fim
```

- **R0**: constante 0
- **R1**: fib1
- **R2**: fib2
- **R3**: temporário



FIM DA UNIDADE 4 (2025/1)