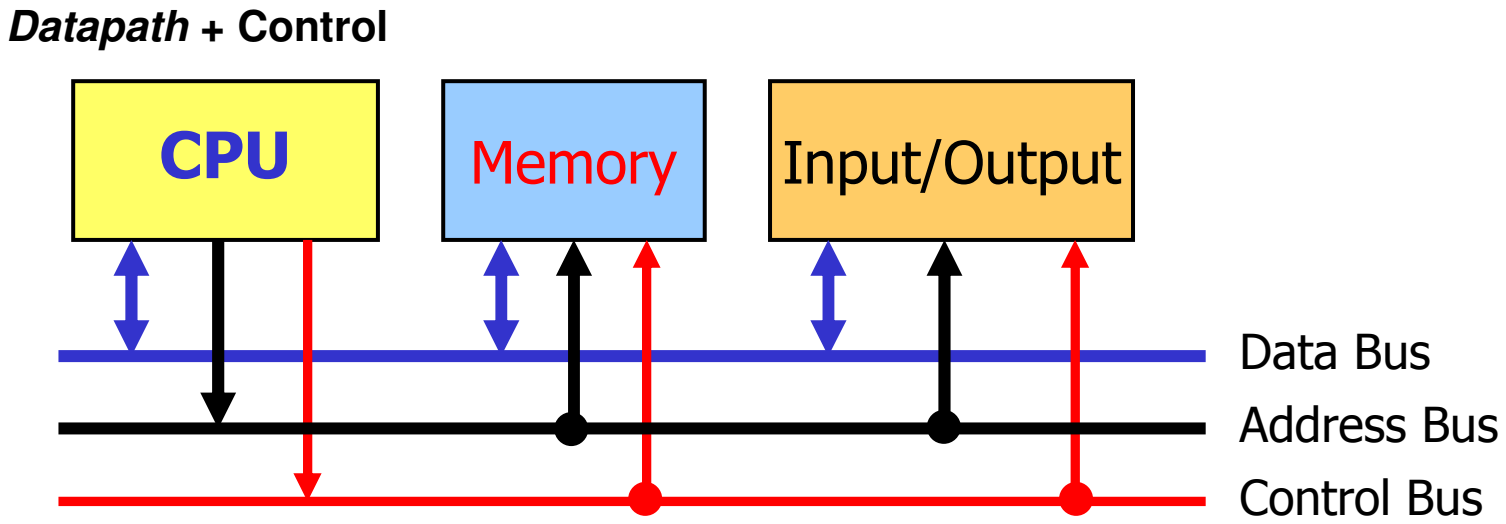
Aulas 14, 15 e 16

- Modelos de Harvard e Von Neumann
- Blocos constituintes de um *datapath* genérico para uma arquitetura tipo MIPS
- Análise dos blocos necessários à execução de um subconjunto de instruções do MIPS, de cada classe de instruções:
  - Aritméticas e lógicas (add, addi, sub, and, or, slt, slti)
  - Acesso à memória (lw, sw)
  - Controlo de fluxo de execução (beq, j)
- Montagem de um *datapath* completo para execução de instruções num único ciclo de relógio (*single-cycle*)

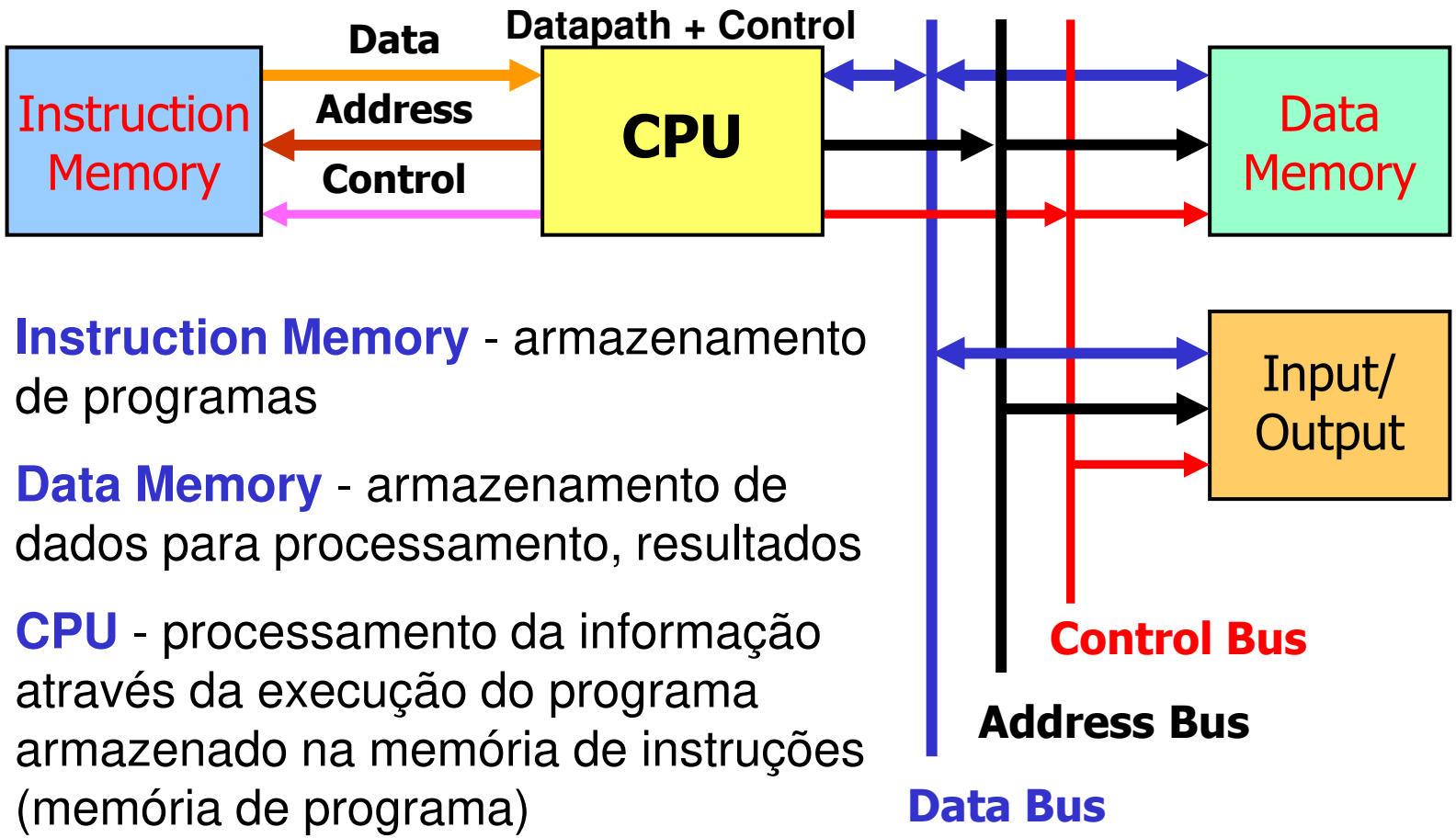
Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Modelo de von Neumann



- **CPU** – processamento da informação através da execução do programa armazenado em memória
- **Memory** – armazenamento de: programas, dados para processamento, resultados
- **Input/Output** – comunicação com o exterior (periféricos)

Modelo de Harvard



- **Instruction Memory** - armazenamento de programas
- **Data Memory** - armazenamento de dados para processamento, resultados
- **CPU** - processamento da informação através da execução do programa armazenado na memória de instruções (memória de programa)

von Neumann *versus* Harvard – resumo

- **Modelo de von Neumann**
  - um único espaço de endereçamento para instruções e dados (i.e. uma única memória)
  - acesso a instruções e dados é feito em ciclos de relógio distintos
- **Modelo de Harvard**
  - dois espaços de endereçamento separados: um para dados e outro para instruções (i.e. duas memórias independentes)
  - possibilidade de acesso, no mesmo ciclo de relógio, a dados e instruções (i.e. CPU pode fazer o *fetch* da instrução e ler os dados que a instrução vai manipular no mesmo ciclo de relógio)
  - memórias de dados e instruções podem ter comprimentos de palavra diferentes

Implementação de um *Datapath*

- O CPU consiste, fundamentalmente, em duas secções:
  - **Secção de dados** (*datapath*) - elementos operativos/funcionais para armazenamento, processamento e encaminhamento da informação:
    - Registos
    - Unidade Aritmética e Lógica (ALU)
    - Elementos de encaminhamento (*multiplexers*)
  - **Unidade de controlo**: responsável pela coordenação dos elementos da secção de dados, durante a execução de cada instrução

Implementação de um *Datapath*

- As unidades funcionais que constituem o *datapath* são de dois tipos:
  - **Elementos combinatórios** (por exemplo a ALU)
  - **Elementos de estado**, isto é, que têm capacidade de armazenamento (por exemplo os registos agrupados num banco de registos, ou outros registos internos) *
- Um elemento de estado possui, pelo menos, duas entradas:
  - Uma para os **dados** a serem armazenados
  - Outra para o **relógio**, que determina o instante em que os dados são armazenados (interface síncrona)
- Um elemento de estado pode ser lido em qualquer momento
- A saída de um elemento de estado disponibiliza a informação armazenada na última transição ativa do relógio

(*) Na abordagem que se faz nestas aulas, e por uma questão de legibilidade dos diagramas, considera-se a memória externa como um elemento operativo integrante do *datapath* (elemento de estado)

Implementação de um *Datapath*

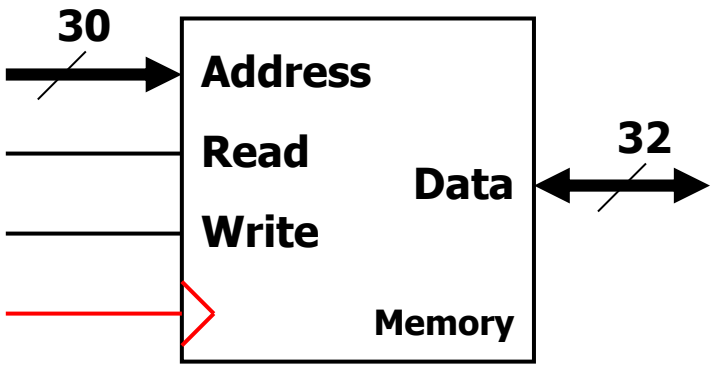
- Para além do sinal de relógio, um elemento de estado pode ainda ter sinais de controlo adicionais:
  - **Um sinal de leitura (read)**, que permite (quando ativo) que a informação armazenada seja disponibilizada na saída (leitura assíncrona)
  - **Um sinal de escrita (write)**, que autoriza (quando ativo) a escrita de informação na próxima transição ativa do relógio (escrita síncrona)
- Se algum destes dois sinais não estiver explicitamente representado, isso significa que a operação respetiva é sempre realizada
  - No caso da operação de escrita ela é realizada uma vez por ciclo, e coincide com a transição ativa do sinal de relógio

NOTA: Nos slides seguintes, por uma questão de simplificação dos diagramas, o sinal de relógio pode não ser sempre explicitamente representado

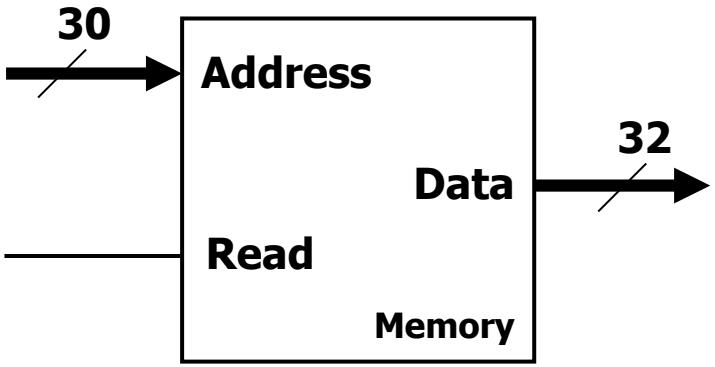
Implementação de um *Datapath*

- Exemplos de representação gráfica de blocos funcionais correspondentes a elementos de estado

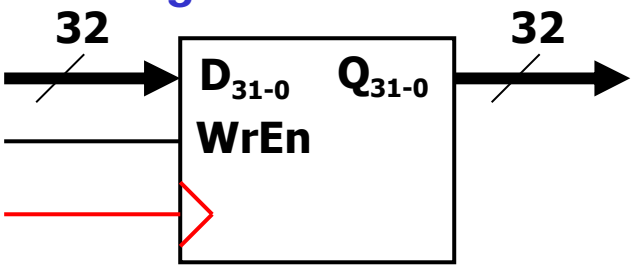
Memória para escrita e leitura (2^{30} words de 32 bits)



Memória apenas para leitura (2^{30} words de 32 bits)



Registo de 32 bits



O sinal "**Read**" pode não existir. Nesse caso a informação de saída estará sempre disponível e corresponderá ao conteúdo da posição de memória especificada na entrada "address"

Implementação de um *Datapath* - MIPS

- Nos próximos slides faz-se uma abordagem à implementação de um *datapath* capaz de interpretar e executar o seguinte subconjunto de instruções do MIPS:
  - As instruções aritméticas e lógicas (**`add`**, **`addi`**, **`sub`**, **`and`**, **`or`**, **`slt`** e **`slti`**)
  - Instruções de acesso à memória: load word (**`lw`**) e store word (**`sw`**)
  - As instruções de salto condicional (**`beq`**) e salto incondicional (**`j`**)
- Independentemente da quantidade e tipo de instruções suportadas por uma dada arquitetura, **uma parte importante do trabalho realizado pelo CPU e da infra-estrutura necessária para executar essas instruções é comum a praticamente todas elas**

Implementação de um *Datapath* - MIPS

- No caso do MIPS, para qualquer instrução que compõe o *set* de instruções, **as duas primeiras operações necessárias à sua execução são sempre as mesmas**:
  1. Usar o conteúdo do registo *Program Counter* (PC) como endereço da memória do qual vai ser lido o código máquina da próxima instrução e efetuar essa leitura
  2. Ler dois registos internos, usando para isso os índices obtidos nos respetivos campos da instrução (`rs` e `rt`):
     - Nas instruções de transferência memória→registo ("`lw`") e nas instruções que operam com constantes (imediatos) apenas o conteúdo de um registo é necessário (codificado no campo `rs`)
     - Em todas as outras é sempre necessário o conteúdo de dois registos (exceto na instrução "`j`")
- **Depois destas operações genéricas, realizam-se as ações específicas para completar a execução da instrução em causa**

Implementação de um *Datapath* - MIPS

- As ações específicas necessárias para executar as instruções de cada uma das três classes de instruções descritas anteriormente são, em grande parte, semelhantes, independentemente da instrução exata em causa
- Por exemplo, **todas as instruções** (à exceção do salto incondicional) **utilizam a ALU depois da leitura dos registos**:
  - as instruções aritméticas e lógicas para a operação correspondente à instrução
  - as instruções de acesso à memória usam a ALU para calcular o endereço de memória
  - a instrução de *branch* para efetuar a subtração que permite determinar se os operandos são iguais ou diferentes
- A execução da instrução de salto incondicional ("`j`") resume-se à alteração incondicional do registo Program Counter (PC) com o endereço-alvo
  - o endereço-alvo é obtido a partir dos 26 LSbits do código máquina da instrução e dos 4 bits mais significativos do valor atual do PC

Implementação de um *Datapath* - MIPS

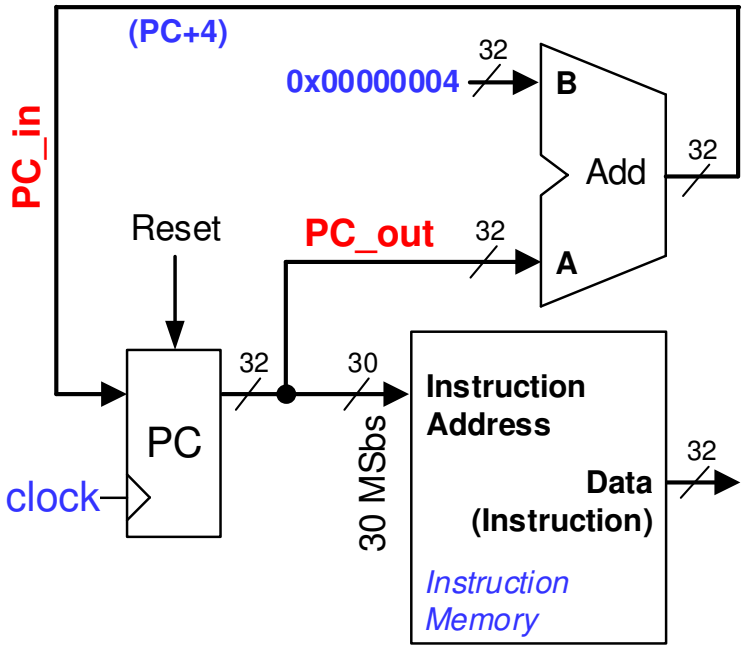
- Depois de utilizar a ALU, as ações que completam as várias classes de instruções diferem:
  - as instruções **aritméticas e lógicas** armazenam o resultado à saída da ALU no registo destino especificado na instrução
  - a instrução "**`sw`**" acede à memória para escrita do valor do registo lido anteriormente (codificado no campo **`rt`**)
  - a instrução "**`lw`**" acede à memória para leitura; o valor lido da memória é, de seguida, escrito no registo destino especificado na instrução (codificado no campo **`rt`**)
  - a instrução "**`beq`**" pode ter que alterar o conteúdo do registo Program Counter (i.e. o endereço onde se encontra a próxima instrução a ser executada) no caso de a condição em teste ser verdadeira

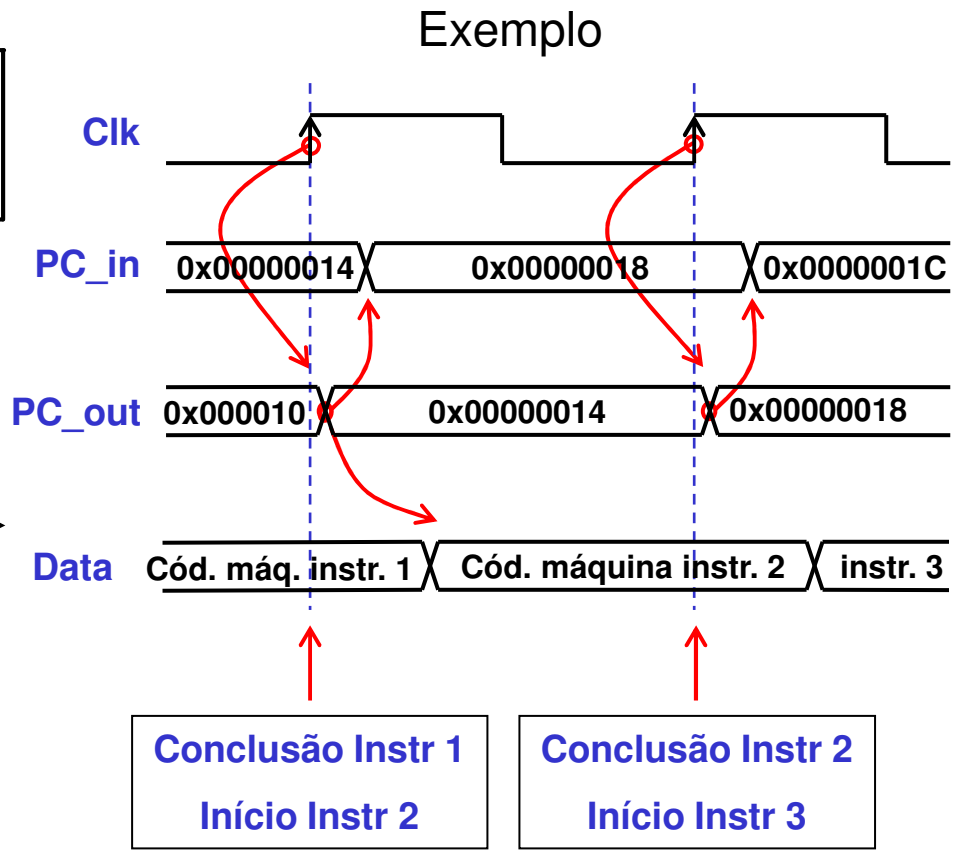
Implementação de um *Datapath* – *Instruction Fetch*

- O processo de acesso à memória para leitura da próxima instrução é genericamente designado por ***Instruction Fetch***
- As instruções que compõem um programa são armazenadas sequencialmente na memória:
  - se a instrução **`n`** se encontra armazenada no endereço **`k`**, então a instrução **`n+1`** encontra-se armazenada no endereço **`k+x`**, em que **`x`** é a dimensão da instrução **`n`**, medida em bytes
  - no MIPS, a dimensão das instruções é fixa e igual a 4 bytes; o endereço **`k`** é sempre um **múltiplo de 4**
- **O processo de *Instruction Fetch* deverá, uma vez concluído, deixar o conteúdo do PC pronto para endereçar a próxima instrução**
  - No caso do MIPS, tal corresponde a adicionar a constante 4 ao valor atual do PC

Implementação de um *Datapath – Instruction Fetch*

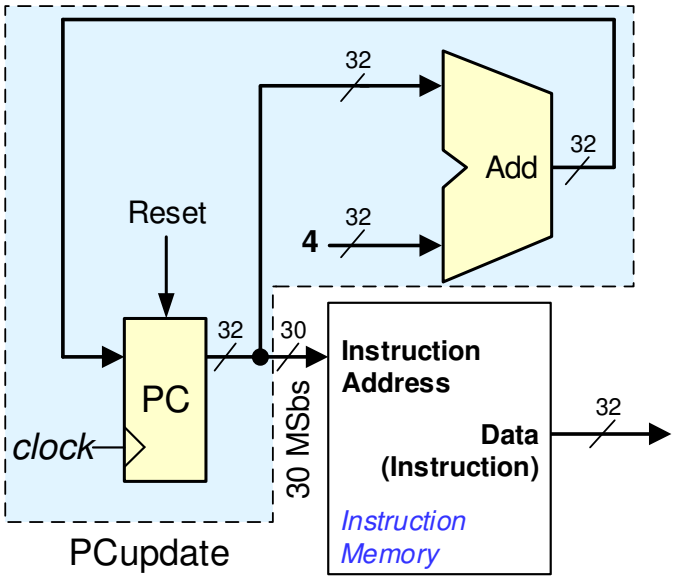
- A parte do *Datapath* necessária à execução de um *Instruction Fetch* toma, assim, a seguinte configuração





Implementação de um *Datapath* – Atualização do PC

```
entity PCupdate is
  port( clk    : in std_logic;
        reset  : in std_logic;
        pc     : out std_logic_vector(31 downto 0));
end PCupdate;

architecture Behavioral of PCupdate is
  signal s_pc : unsigned(31 downto 0);
begin
  process(clk)
  begin
     if(rising_edge(clk)) then
        if(reset = '1') then
           s_pc <= (others => '0');
        else
           s_pc <= s_pc + 4;
        end if;
     end if;
  end process;
  pc <= std_logic_vector(s_pc);
end Behavioral;
```



Implementação de um *Datapath – Instruction Memory*

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

entity InstructionMemory is
  generic(ADDR_BUS_SIZE : positive := 6);
  port( address  : in  std_logic_vector(ADDR_BUS_SIZE-1 downto 0);
        readData : out std_logic_vector(31 downto 0));
end InstructionMemory;

architecture Behavioral of InstructionMemory is
  constant NUM_WORDS : positive := (2 ** ADDR_BUS_SIZE );
  subtype TData is std_logic_vector(31 downto 0);
  type TMemory is array(0 to NUM_WORDS - 1) of TData;
  constant s_memory : TMemory :=  (X"8C610004",   -- lw   $1,4($3)
                                   X"20210004",   -- addi $1,$1,4
                                   X"AC610008",   -- sw   $1,8($0)
                                   others => X"00000000");
begin
  readData <= s_memory(to_integer(unsigned(address)));
end Behavioral;
```

Implementação de um *Datapath*

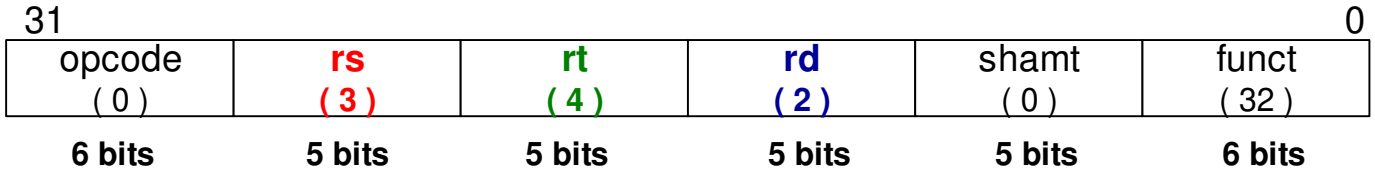
- Que outros elementos operativos básicos serão necessários para suportar a execução das várias classes de instruções que estamos a considerar?
  - Instruções aritméticas e lógicas
    - Tipo R: **`add`**, **`sub`**, **`and`**, **`or`**, **`slt`**
    - Tipo I: **`addi`**, **`slti`**
  - Instruções de leitura e escrita da memória (Tipo I: **`lw`**, **`sw`**)
  - Instrução de salto condicional (Tipo I: **`beq`**)

Na análise que se segue, não se explicita a Unidade de Controlo. Esta unidade é responsável pela geração dos sinais de controlo que asseguram a coordenação dos elementos do *datapath* durante a execução de uma instrução

Implementação de um *Datapath* – instruções tipo R

- Operações realizadas no decurso da execução de uma instrução tipo R:
  - ***Instruction Fetch*** (leitura da instrução, cálculo de PC+4)
  - **Leitura dos registos** operando (registos especificados nos campos "`rs`" e "`rt`" da instrução)
  - **Realização da operação** na ALU (especificada no campo "`funct`")
  - **Escrita do resultado** no registo destino (especificado no campo "`rd`")

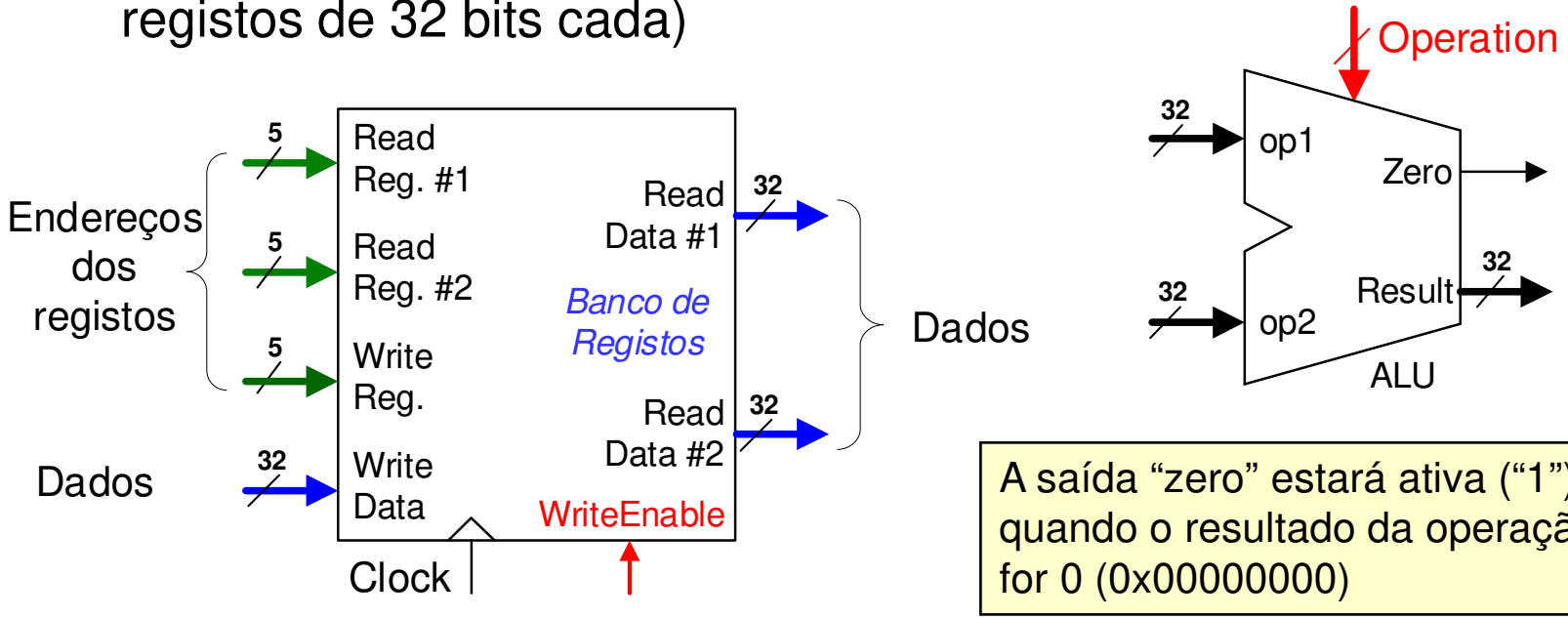
Exemplo: **add $2, $3, $4**



Código máquina: 0x00641020

Implementação de um *Datapath* – instruções tipo R

- Os elementos necessários à execução das instruções aritméticas e lógicas (tipo R) são:
  - Uma ALU de 32 bits
  - Um conjunto de registos internos (Banco de registos com 32 registos de 32 bits cada)

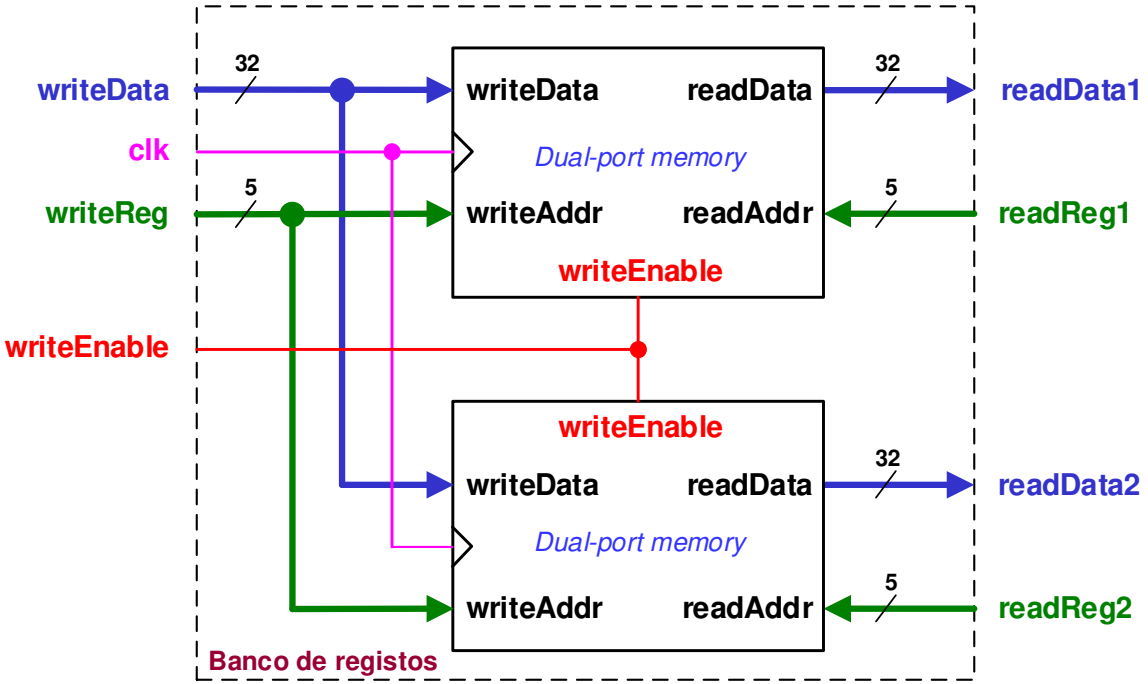


A saída "zero" estará ativa ("1") quando o resultado da operação for 0 (0x00000000)

  - 2 portos de leitura assíncrona
  - 1 porto de escrita síncrona

Banco de Registos

- O banco de registos pode ser implementado com duas memórias de duplo porto (um porto de escrita e um porto de leitura):



  - o porto de escrita do banco de registos é comum às duas memórias (i.e. a escrita é feita simultaneamente nas duas memórias)
  - cada memória fornece um porto de leitura independente

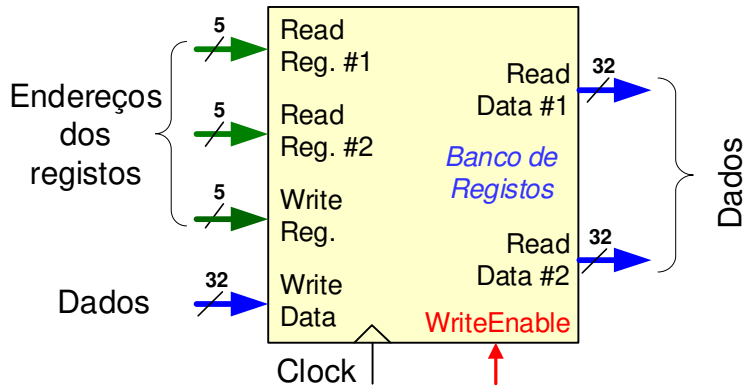
Banco de registos (dual-port memory) – VHDL

```
entity DP_Memory is
  generic(WORD_BITS  : integer range 1 to 128 := 32;
          ADDR_BITS  : integer range 1 to  10 := 5);
  port(
    clk        : in  std_logic;
    -- asynchronous read port
    readAddr  : in std_logic_vector(ADDR_BITS-1 downto 0);
    readData  : out std_logic_vector(WORD_BITS-1 downto 0);

    -- synchronous write port
    writeAddr: in std_logic_vector(ADDR_BITS-1 downto 0);
    writeData: in std_logic_vector(WORD_BITS-1 downto 0);
    writeEnable : in std_logic);
end DP_Memory;
```

Banco de registos (dual-port memory) – VHDL

```
architecture Behavioral of DP_Memory is
  subtype TDataWord is std_logic_vector(WORD_BITS-1 downto 0);
  type TMem is array (0 to 2**ADDR_BITS-1) of TDataWord;
  signal s_memory : TMem := (others => (others => '0'));
begin
  process(clk, writeEnable) is
  begin
     if(rising_edge(clk) ) then
       if(writeEnable = '1') then
         s_memory(to_integer(unsigned(writeAddr))) <= writeData;
       end if;
     end if;
  end process;
  readData <= (others => '0') when
                    (to_integer(unsigned(readAddr)) = 0) else
                     s_memory(to_integer(unsigned(readAddr)));
end Behavioral;
```

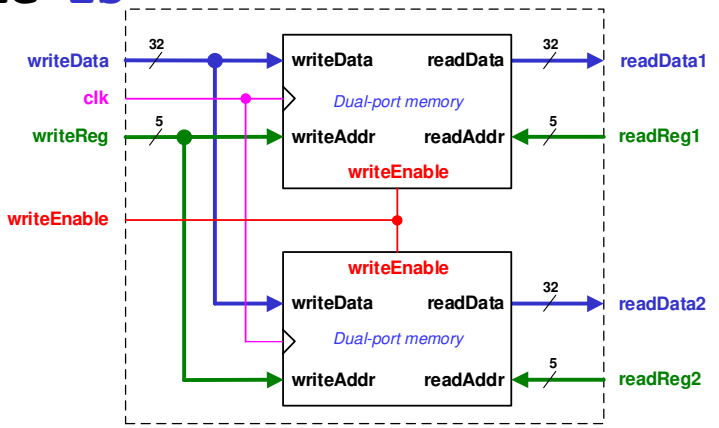
Banco de registos – VHDL




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

entity RegFile is
 port(clk   : in  std_logic;
    -- synchronous write port
       writeEnable   : in  std_logic;
       writeReg      : in  std_logic_vector( 4 downto 0);
       writeData     : in  std_logic_vector(31 downto 0);
    -- asynchronous read port #1
       readReg1      : in  std_logic_vector( 4 downto 0);
       readData1     : out std_logic_vector(31 downto 0);
    -- asynchronous read port #2
       readReg2      : in  std_logic_vector( 4 downto 0);
       readData2     : out std_logic_vector(31 downto 0));
end RegFile;
```

Banco de registos – VHDL

```
architecture Structural of RegFile is
begin
rs_mem:
entity work.DP_Memory(Behavioral)
  port map(clk         => clk,
           readAddr    => readReg1,
           readData    => readData1,
           writeAddr   => writeReg,
           writeData   => writeData,
           writeEnable => writeEnable);
rt_mem:
entity work.DP_Memory(Behavioral)
  port map(clk         => clk,
           readAddr    => readReg2,
           readData    => readData2,
           writeAddr   => writeReg,
           writeData   => writeData,
           writeEnable => writeEnable);
end Structural;
```



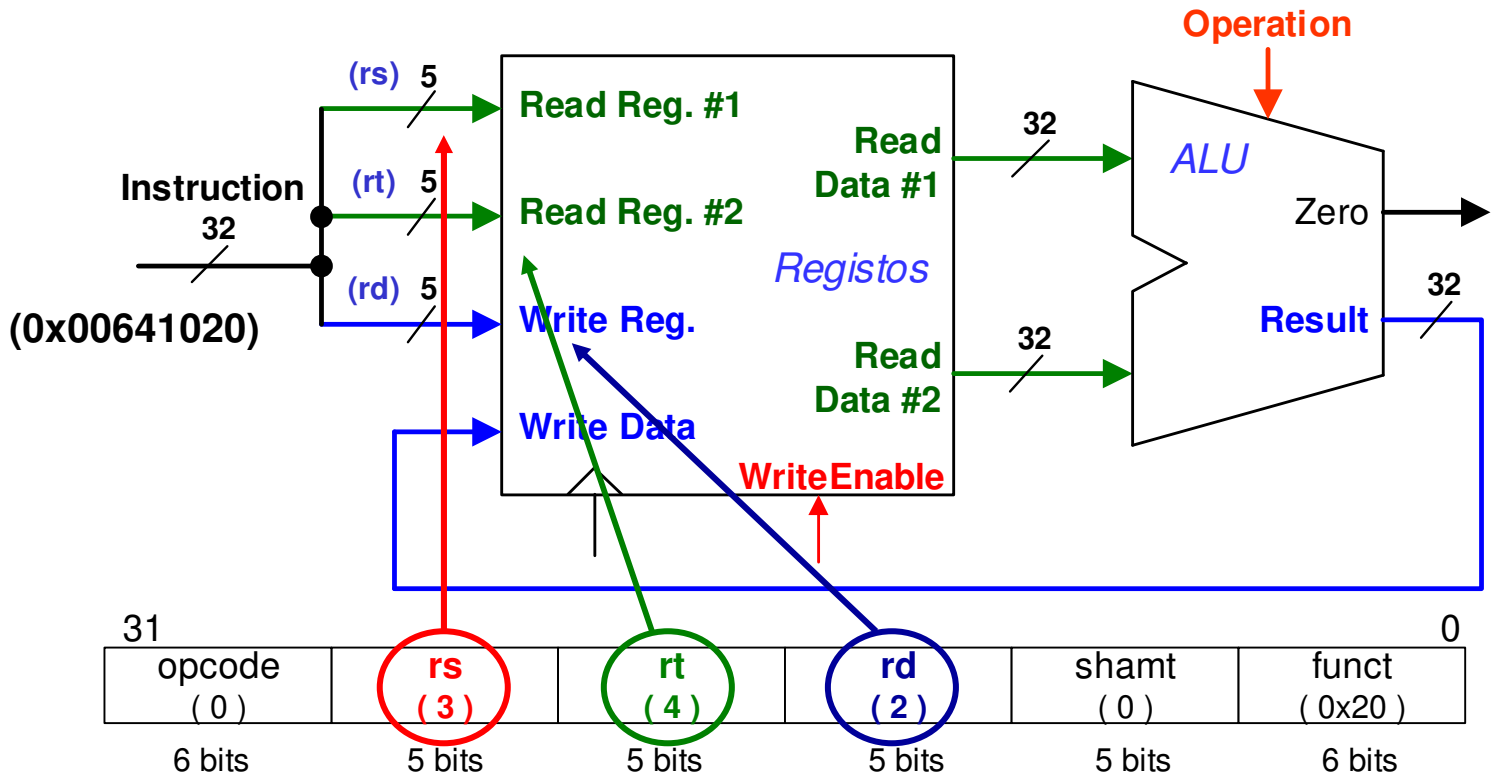

```
entity RegFile is
 port(  clk
        writeEnable
        writeReg
        writeData
        readReg1
        readData1
        readReg2
        readData2
end RegFile;
```

Implementação de um *Datapath* – instruções tipo R

- Interligação dos elementos operativos para a execução de uma instrução tipo R:

Ex.: **add $2, $3, $4**

00000000011001000001000000100000



Implementação de um *Datapath* (Instrução SW)

- Operações realizadas na execução de uma instrução "**`sw`**":
  - *Instruction Fetch* (leitura da instrução, cálculo de PC+4)
  - Leitura dos registos que contêm o **endereço-base** e o **valor a transferir** (registos especificados nos campos "**`rs`**" e "**`rt`**"da instrução, respetivamente)
  - Cálculo, na ALU, do endereço de acesso (soma algébrica entre o conteúdo do registo "**`rs`**" e o **`offset`** especificado na instrução)
  - Escrita na memória

Exemplo: **sw $2, 0x24($4)**

Endereço inicial da memória onde vai ser escrita a word de 32 bits armazenada no registo $2

Implementação de um *Datapath* (Instrução LW)

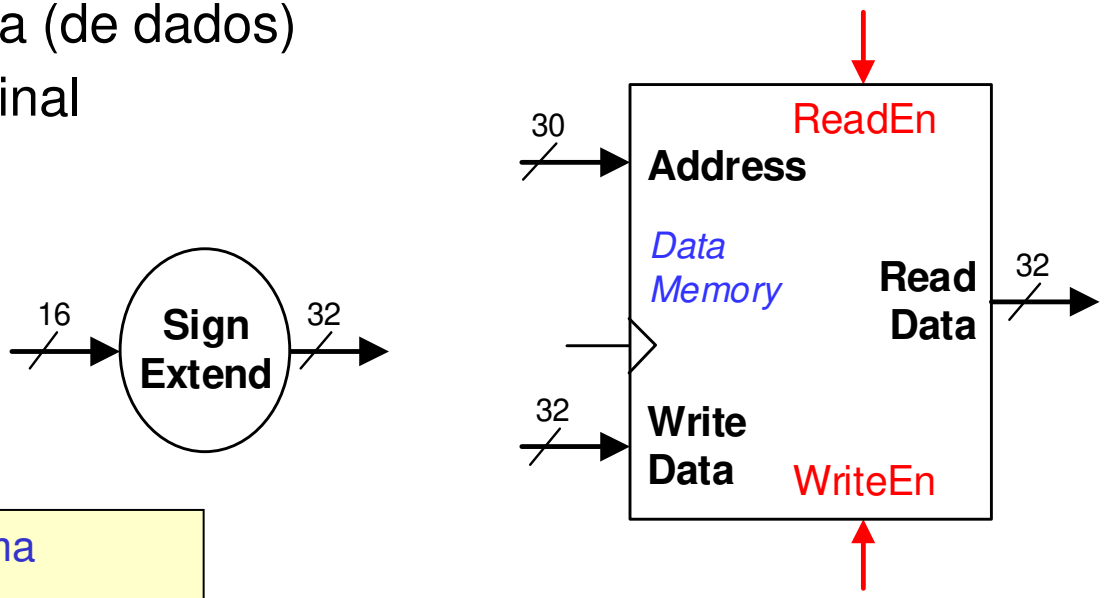
- Operações realizadas na execução de uma instrução "**`lw`**"
  - *Instruction Fetch* (leitura da instrução, cálculo de PC+4)
  - Leitura do registo que contém o endereço base (registo especificado no campo "**`rs`**" da instrução)
  - Cálculo, na ALU, do endereço de acesso (soma algébrica entre o conteúdo do registo "**`rs`**" e o **`offset`** especificado na instrução)
  - Leitura da memória
  - Escrita do valor lido da memória no registo destino (especificado no campo "**`rt`**" da instrução)

Exemplo: **lw $4, 0x2F($15)**

Endereço inicial da memória para leitura de uma word de 32 bits (vai ser escrita no registo $4)

Implementação de um *Datapath* (Instruções lw e sw)

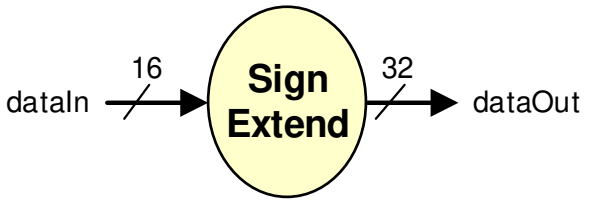
- Os elementos necessários à execução das instruções de transferência de informação entre registos e memória (*load* e *store*) são, para além da ALU e do Banco de Registos:
  - A memória externa (de dados)
  - Um extensor de sinal



O **extensor de sinal** cria uma constante de 32 bits em complemento para 2, a partir dos 16 bits menos significativos da instrução (o bit 15 é replicado nos 16 mais significativos da constante de saída)

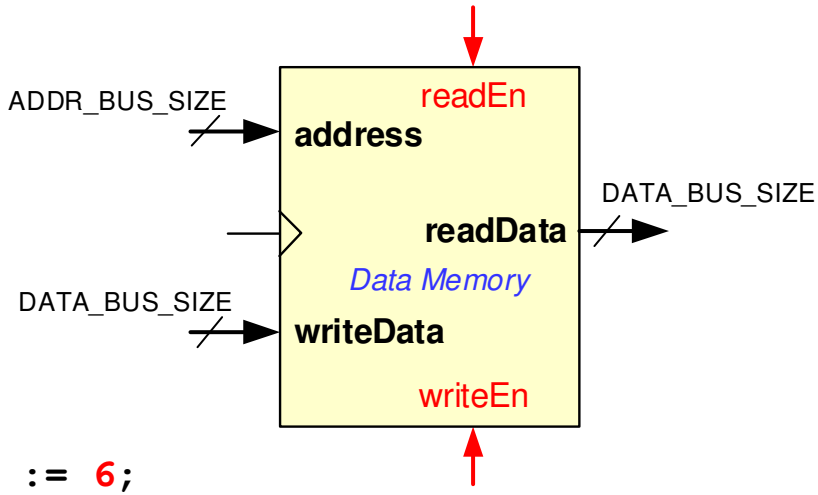
Por uma questão de conveniência de desenho dos diagramas, o barramento de dados da memória (bidirecional) está separado em dados para escrita e dados de leitura

Módulo de extensão de sinal – VHDL




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

entity SignExtend is
 port(dataIn  : in  std_logic_vector(15 downto 0);
      dataOut : out std_logic_vector(31 downto 0));
end SignExtend;

architecture Behavioral of SignExtend is
begin
 dataOut(31 downto 16)  <= (others => dataIn(15));
 dataOut(15 downto 0)   <= dataIn;
end Behavioral;
```

Módulo de memória RAM – VHDL




```
entity RAM is
  generic(ADDR_BUS_SIZE : positive := 6;
          DATA_BUS_SIZE : positive := 32);
  port(clk         : in  std_logic;
       readEn      : in  std_logic;
       writeEn     : in  std_logic;
       address     : in  std_logic_vector(ADDR_BUS_SIZE-1 downto 0);
       writeData   : in  std_logic_vector(DATA_BUS_SIZE-1 downto 0);
       readData    : out std_logic_vector(DATA_BUS_SIZE-1 downto 0));
end RAM;
```

Módulo de memória RAM – VHDL

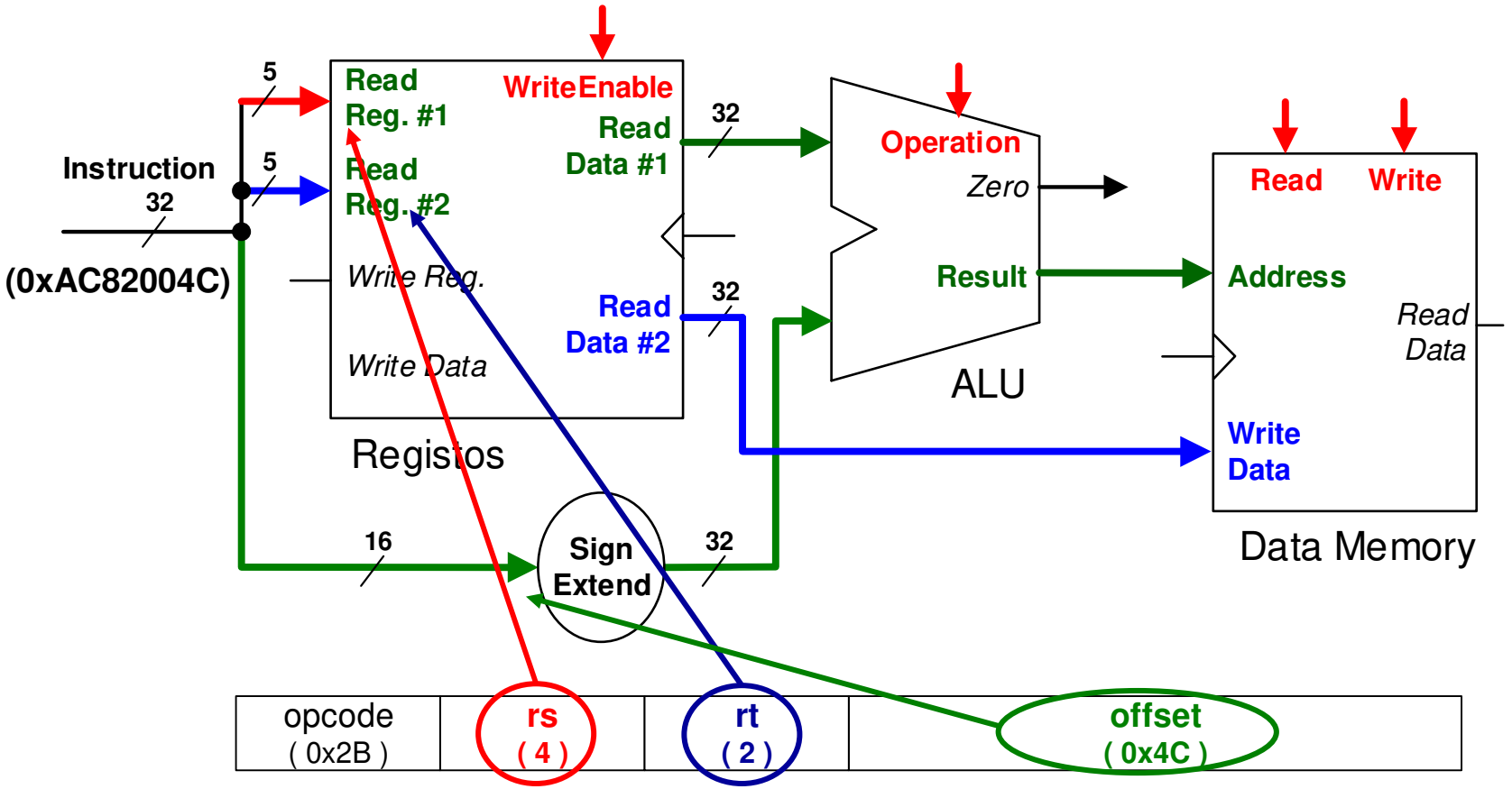
```
architecture Behavioral of RAM is
  constant NUM_WORDS : positive := (2 ** ADDR_BUS_SIZE );
  subtype TData is std_logic_vector(DATA_BUS_SIZE-1 downto 0);
  type TMemory is array(0 to NUM_WORDS - 1) of TData;
  signal s_memory : TMemory;
begin

  process(clk)
  begin
     if(rising_edge(clk)) then
        if(writeEn = '1') then
           s_memory(to_integer(unsigned(address))) <= writeData;
        end if;
     end if;
  end process;
  readData <= s_memory(to_integer(unsigned(address))) when
                  readEn = '1' else (others => 'Z');
end Behavioral;
```

Implementação de um *Datapath* (Instruções lw e sw)

- Interligação dos elementos operativos para a execução do "**sw**":

Ex: **sw $2, 0x4C($4)**

10101100100000010000000000001001100

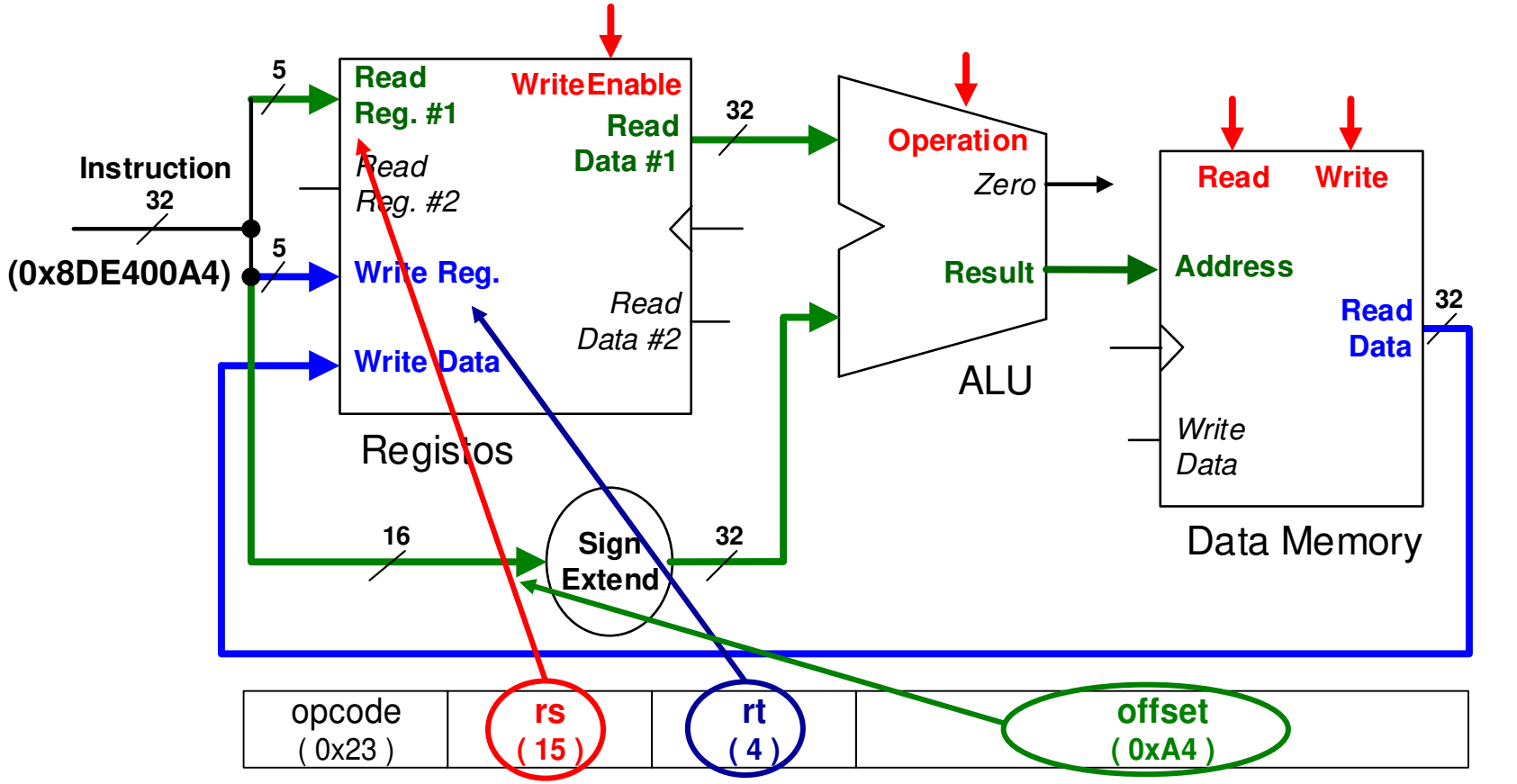


Implementação de um *Datapath* (Instruções lw e sw)

- Interligação dos elementos operativos para a execução do "**lw**":

Ex: **lw $4, 0xA4($15)**

10001101111100100000000000010100100

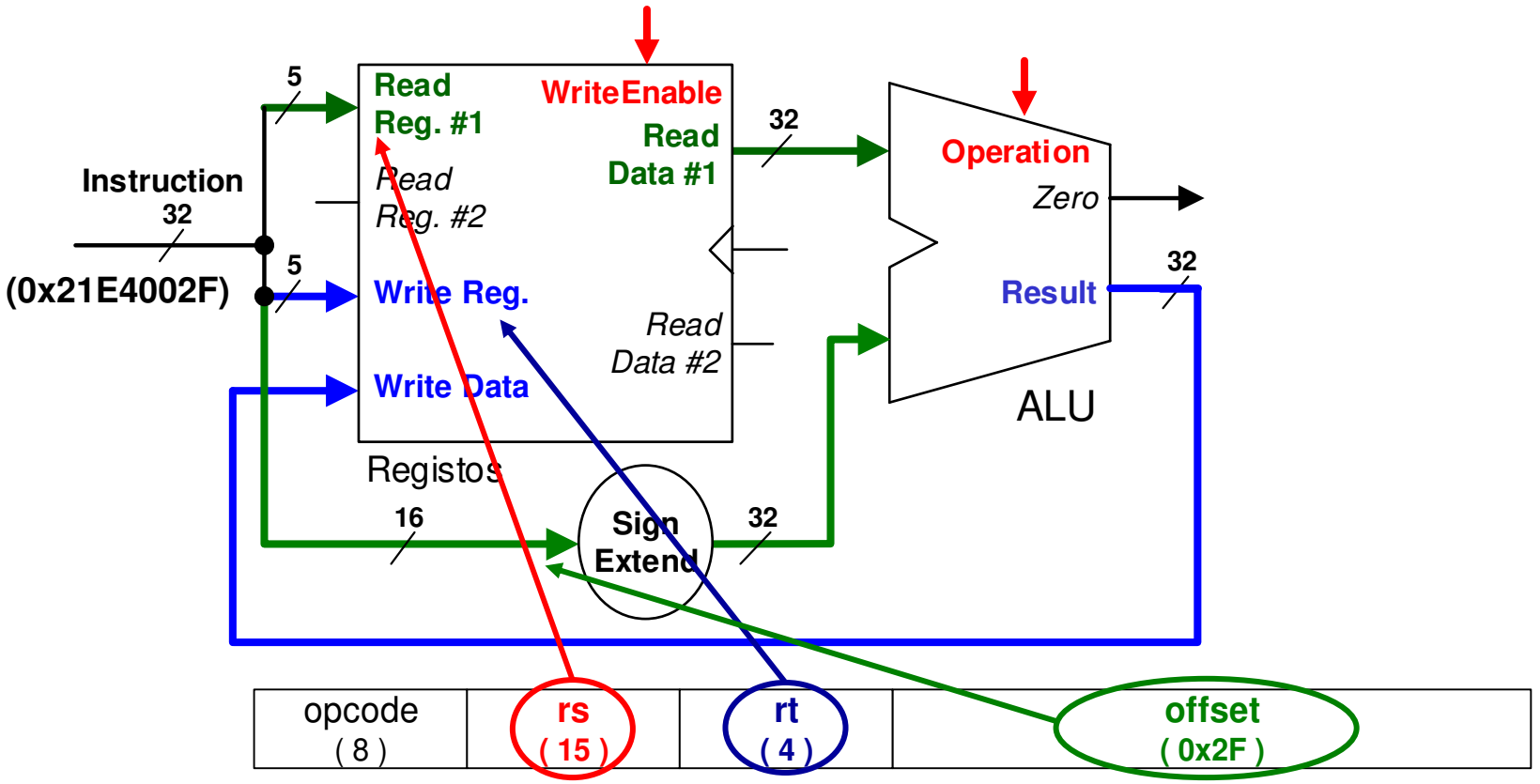


Implementação de um *Datapath* (Instruções "imediatas")

- Interligação dos elementos operativos necessários à execução de instruções que operam com constantes ("**addi**", "**slti**"):

Ex: **addi $4, $15, 0x2F**

00100001111001000000000000101111



Implementação de um *Datapath* (Instruções de *branch*)

- Operações realizadas na execução de uma instrução de *branch*:
  - *Instruction Fetch* (leitura da instrução, cálculo de PC+4)
  - Leitura de dois registos, do banco de registos
  - Comparação dos conteúdos dos registos (realização de uma operação de subtração na ALU)
  - Cálculo do endereço-alvo da instrução de *branch* (*Branch Target Address* - BTA)

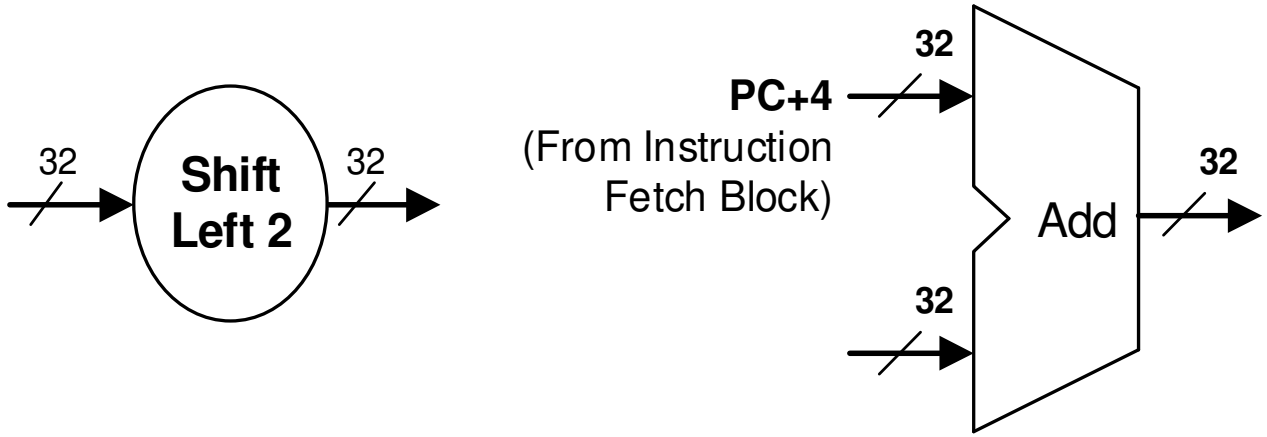
    `BTA = (PC+4) + (instruction_offset << 2)`
  - Alteração do valor do registo PC:
    - se a condição testada pelo *branch* for verdadeira PC = BTA
    - se a condição testada pelo *branch* for falsa PC = PC + 4

Exemplo: **beq $2, $3, 0x20**

opcode (4)	**rs (2)**	**rt (3)**	**instruction_offset (0x20)**

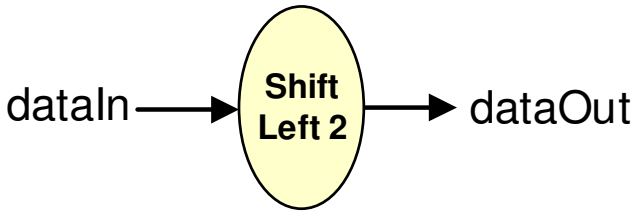
Implementação de um *Datapath* (Instruções de *branch*)

- Os elementos necessários à execução das instruções de salto condicional implicam a inclusão dos seguintes elementos:
  - *left shifter* (2 bits)
  - um somador



O *left shifter* recupera os 2 bits menos significativos da diferença de endereços que são desprezados no momento da codificação da instrução

Módulo "left shifter" – VHDL



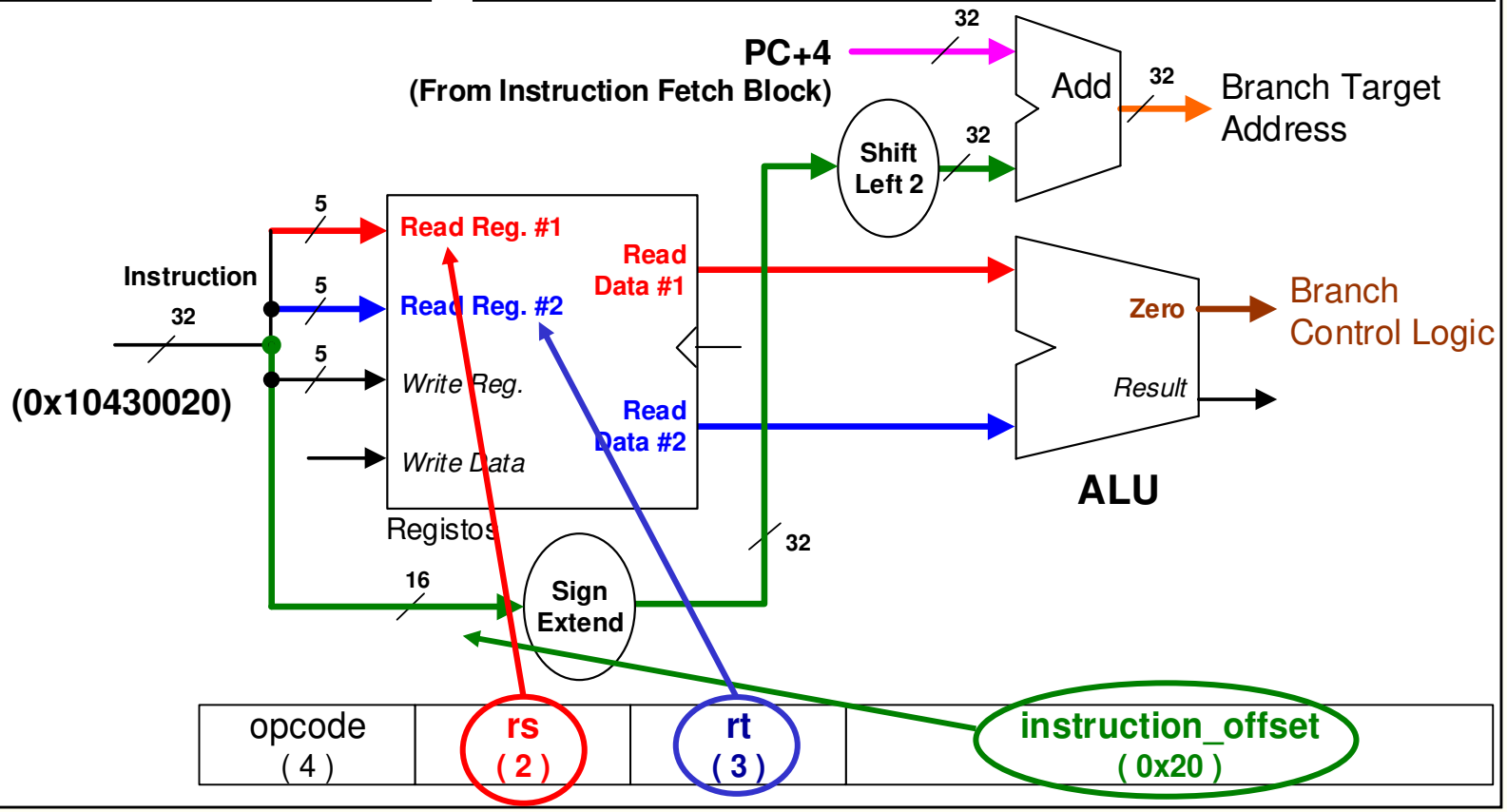

```
library ieee;
use ieee.std_logic_1164.all;

entity LeftShifter2 is
 port(dataIn : in  std_logic_vector(31 downto 0);
      dataOut: out std_logic_vector(31 downto 0));
end LeftShifter2;

architecture Behavioral of LeftShifter2 is
begin
 dataOut <= dataIn(29 downto 0) & "00";
end Behavioral;
```

Implementação de um *Datapath* (Instruções de *branch*)

- Interligação dos elementos operativos necessários à execução de uma instrução de ***branch***:

Ex.: **beq $2, $3, 0x20**

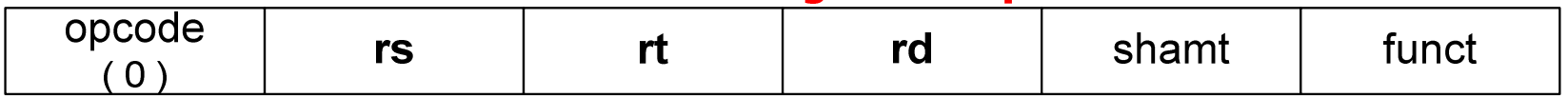
00010000010000110000000000100000



Implementação de um *Datapath* – juntando tudo

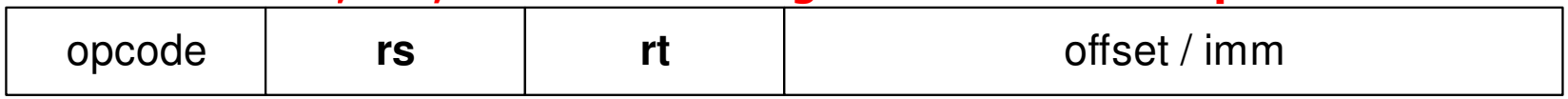
- Relembremos o formato de codificação dos três tipos de instruções:

Aritméticas e lógicas – Tipo R



6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits

LW, SW, aritméticas e lógicas imediatas – Tipo I



6 bits | 5 bits | 5 bits | 16 bits

Branches – Tipo I

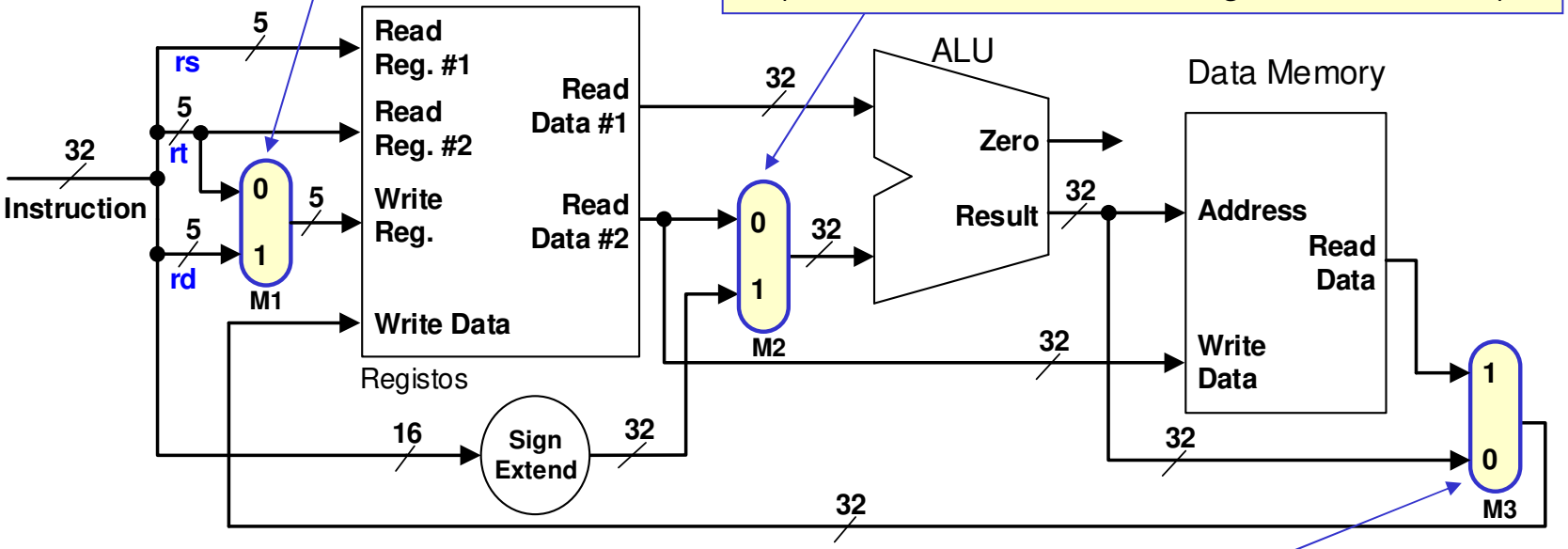
6 bits | 5 bits | 5 bits | 16 bits

Implementação de um *Datapath* – juntando tudo

- **1º passo**: combinação das instruções de acesso à memória com as instruções aritméticas e lógicas do tipo R e do tipo I:

Seleção do registo destino: **rd** (instruções tipo R), **rt** (LW e nas aritméticas e lógicas imediatas)

Seleção do 2º operando da ALU: **conteúdo de um registo** (instruções tipo R e branches); **offset estendido para 32 bits** (LW, SW, aritméticas e lógicas imediatas)

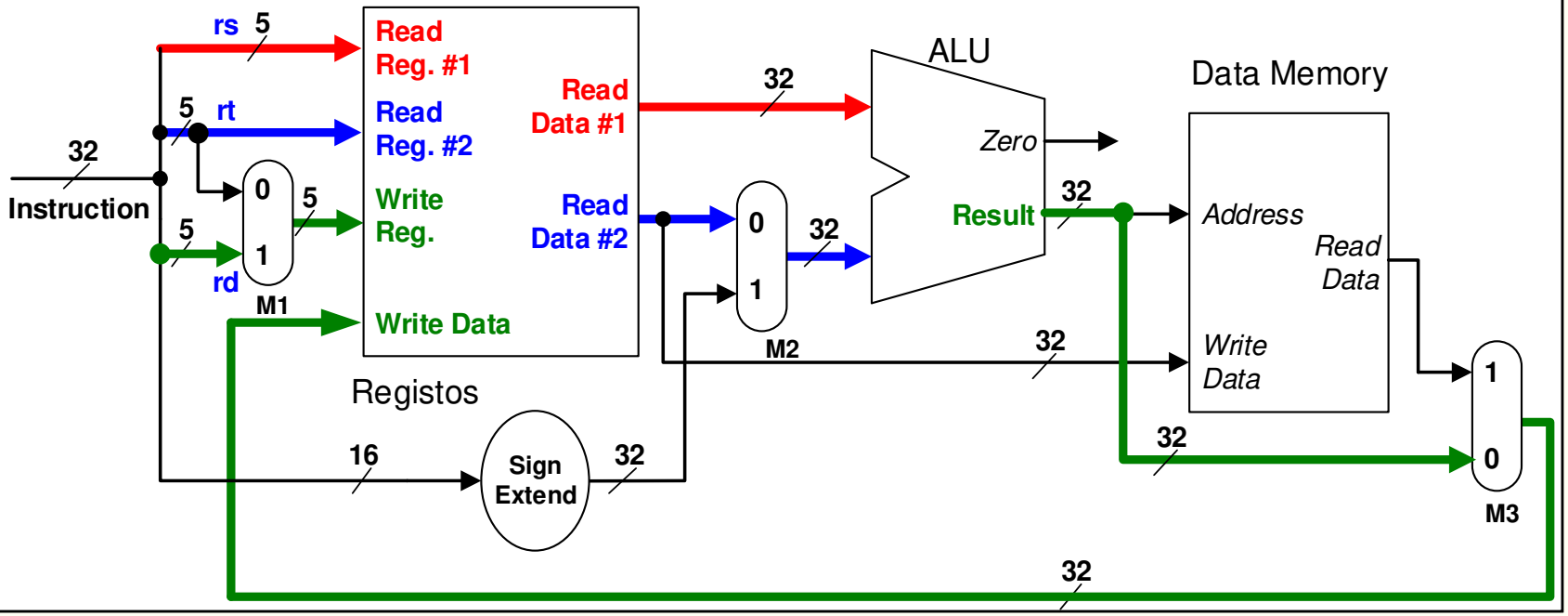


Seleção do valor a escrever no banco de registos: **valor lido da memória** (LW), **valor calculado na ALU** (instruções tipo R, aritméticas e lógicas imediatas)

Implementação de um *Datapath* – juntando tudo

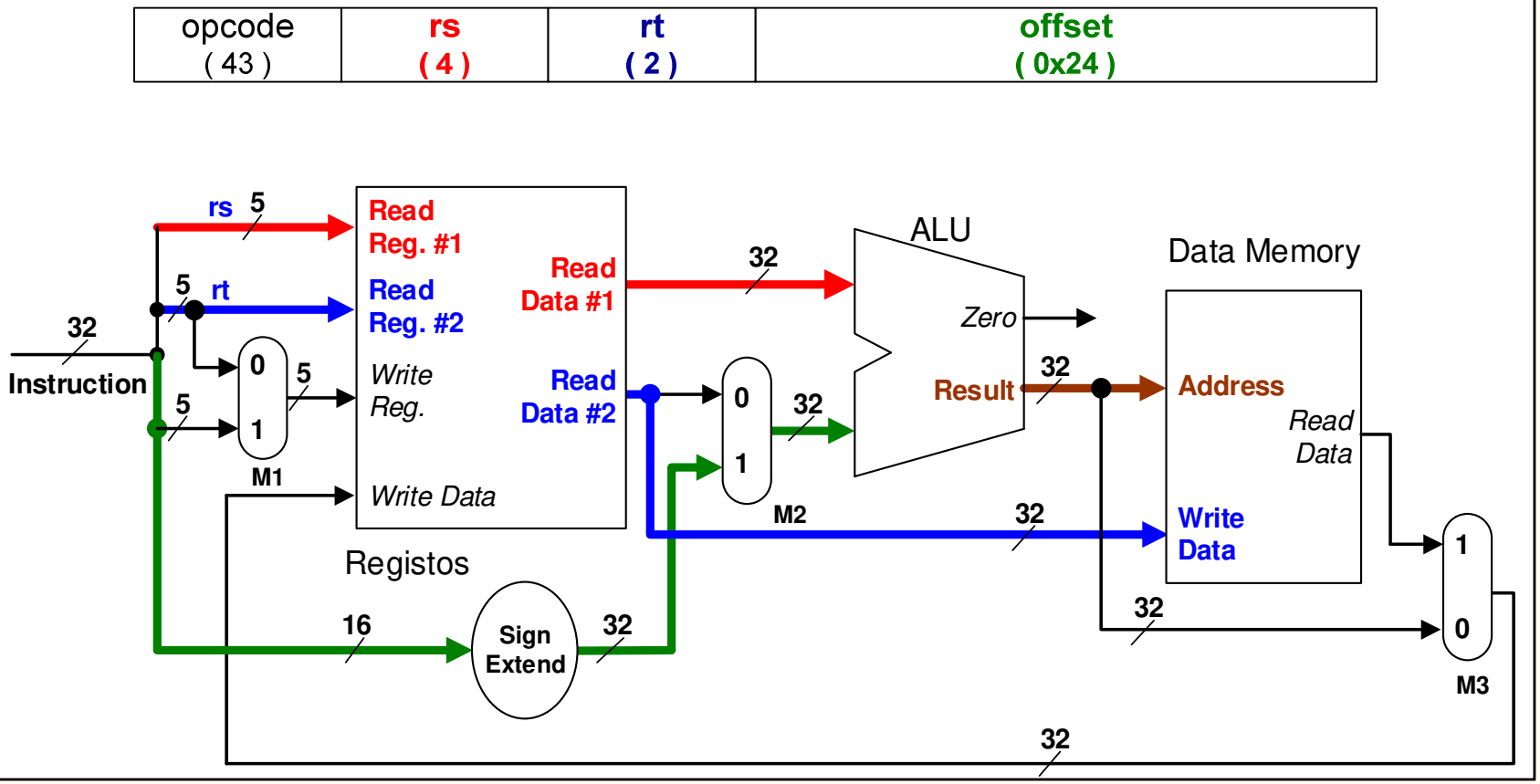
- Fluxo da informação na execução de uma instrução do tipo R. Exemplo: **`add $2,$3,$4`**

opcode	rs	rt	rd	shamt	funct
(0)	(3)	(4)	(2)	(0)	(32)



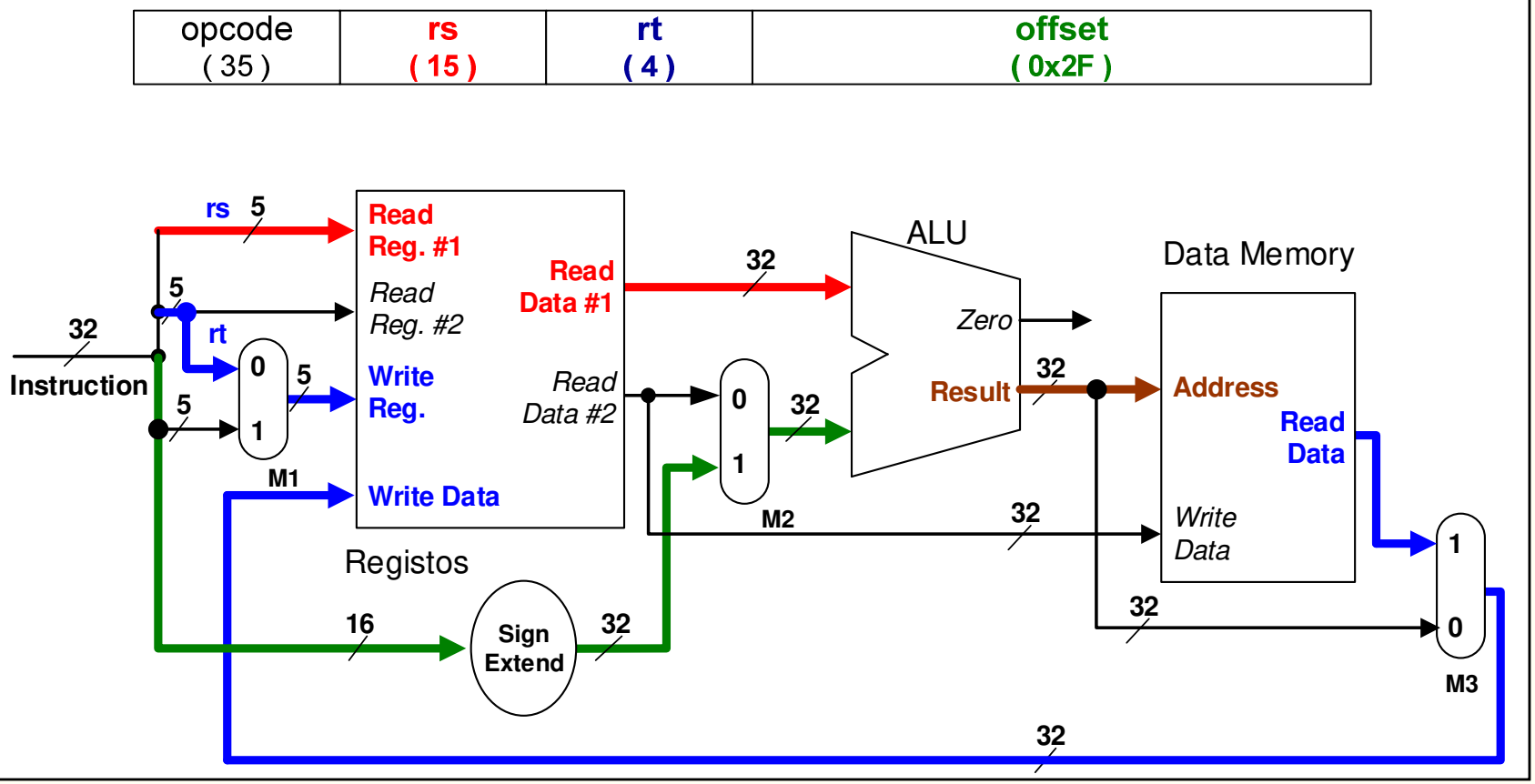
Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução SW (*store word*). Exemplo: `sw $2,0x24($4)`



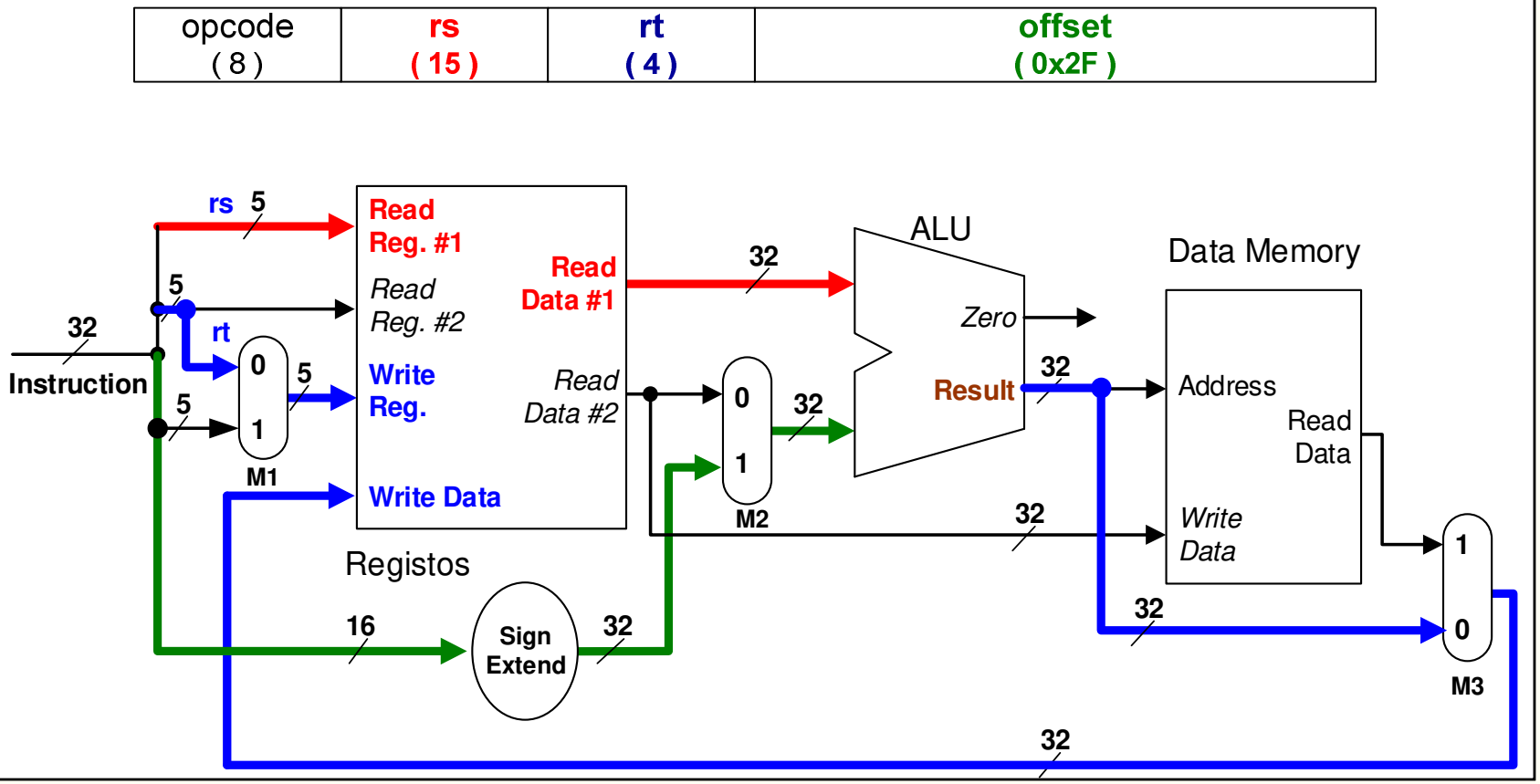
Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução LW (*load word*). Exemplo: `lw $4,0x2F($15)`



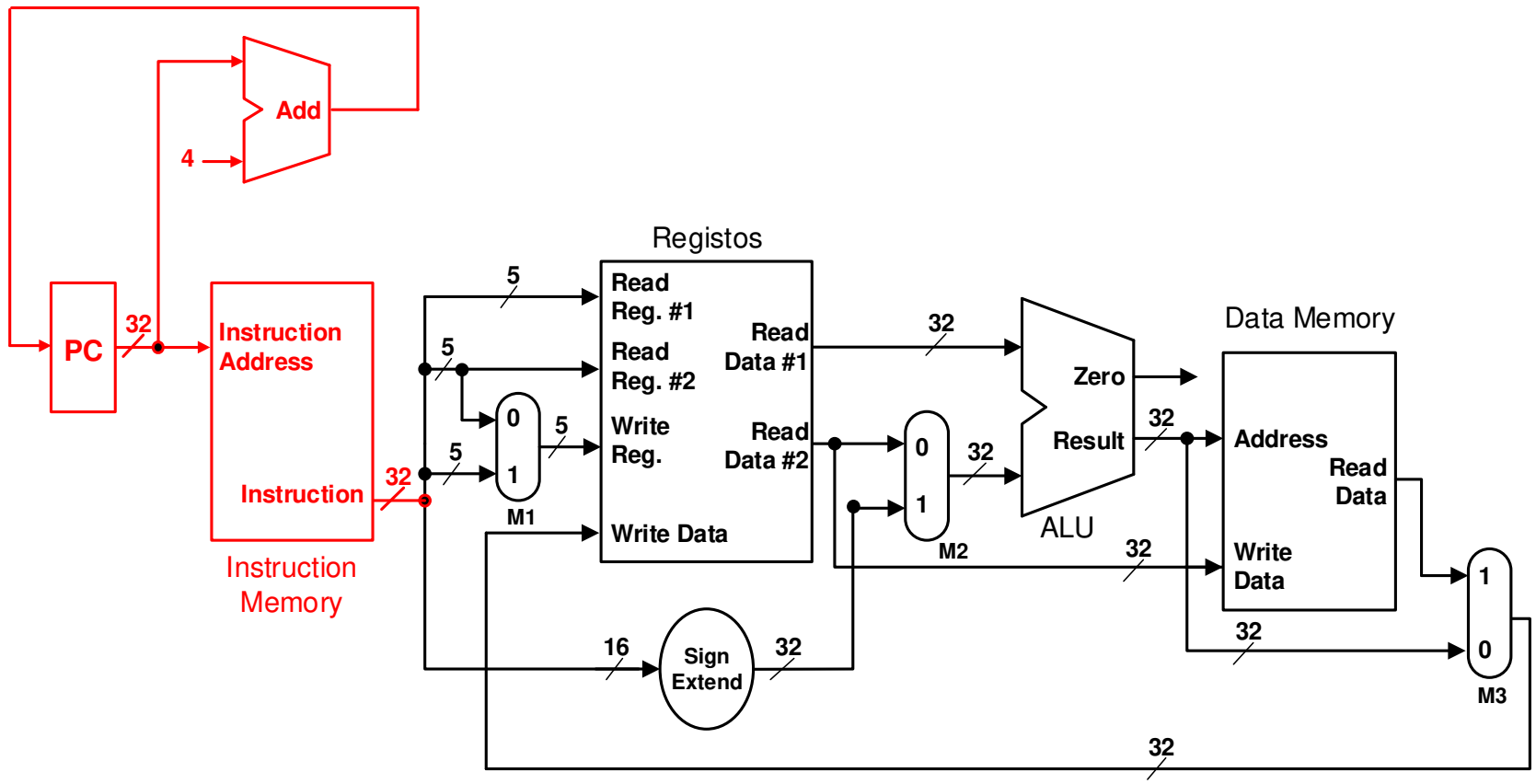
Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução das instruções imediatas.
  Exemplo: **`addi $4,$15,0x2F`**



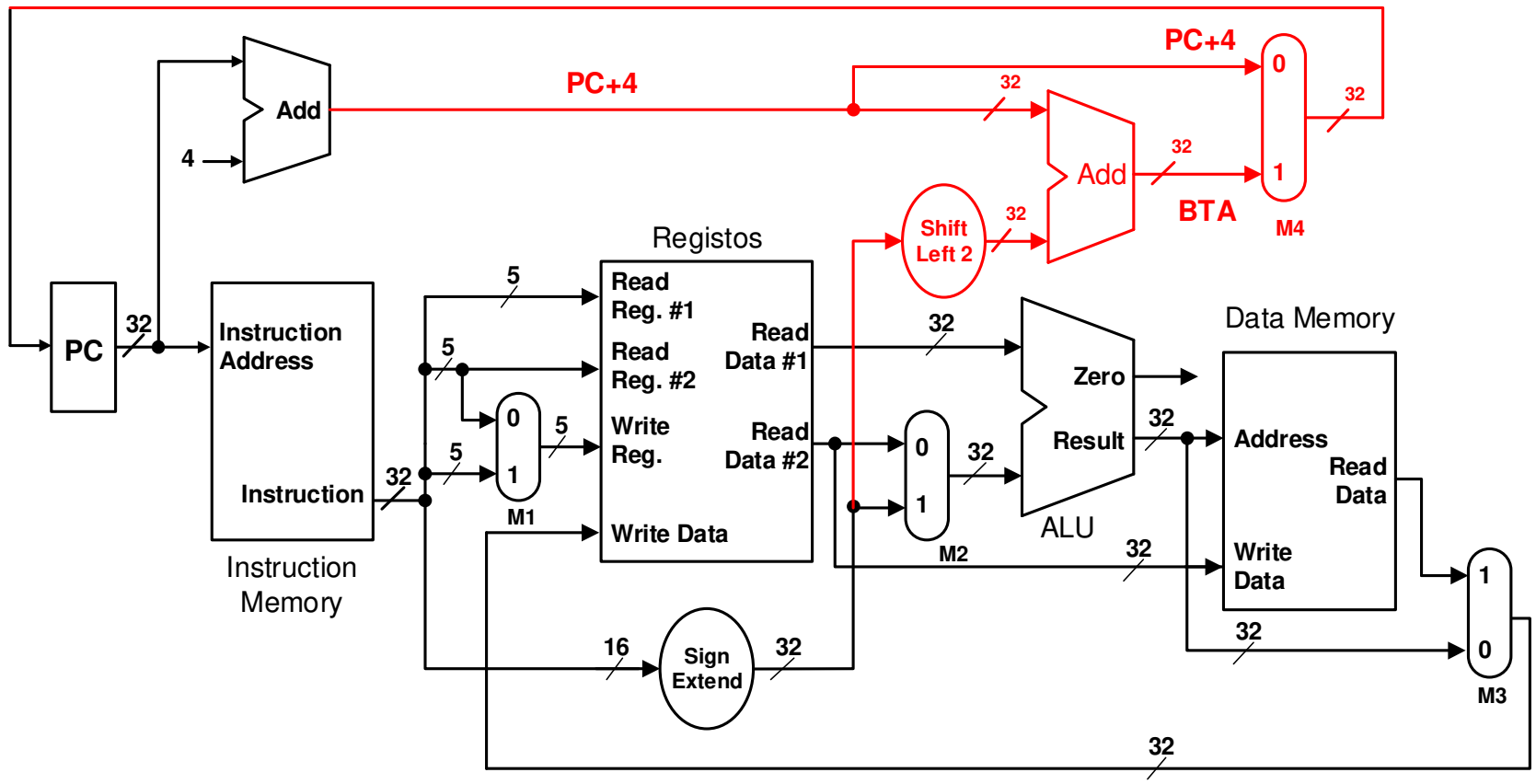
Implementação de um *Datapath* – juntando tudo

- **2º passo**: inclusão do bloco *Instruction Fetch*



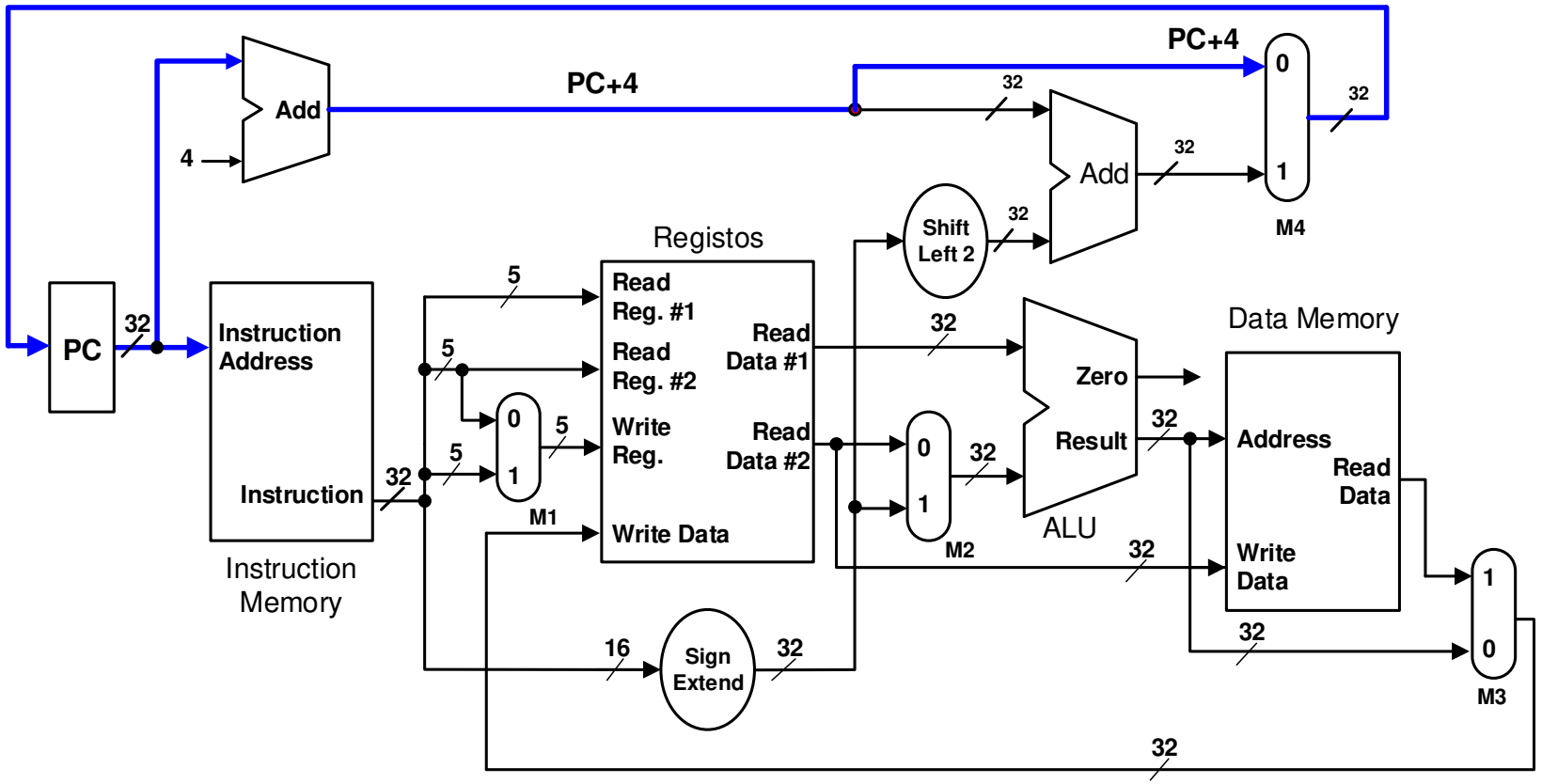
Implementação de um *Datapath* – juntando tudo

- **3º passo**: adição das instruções de salto condicional (*branches*)



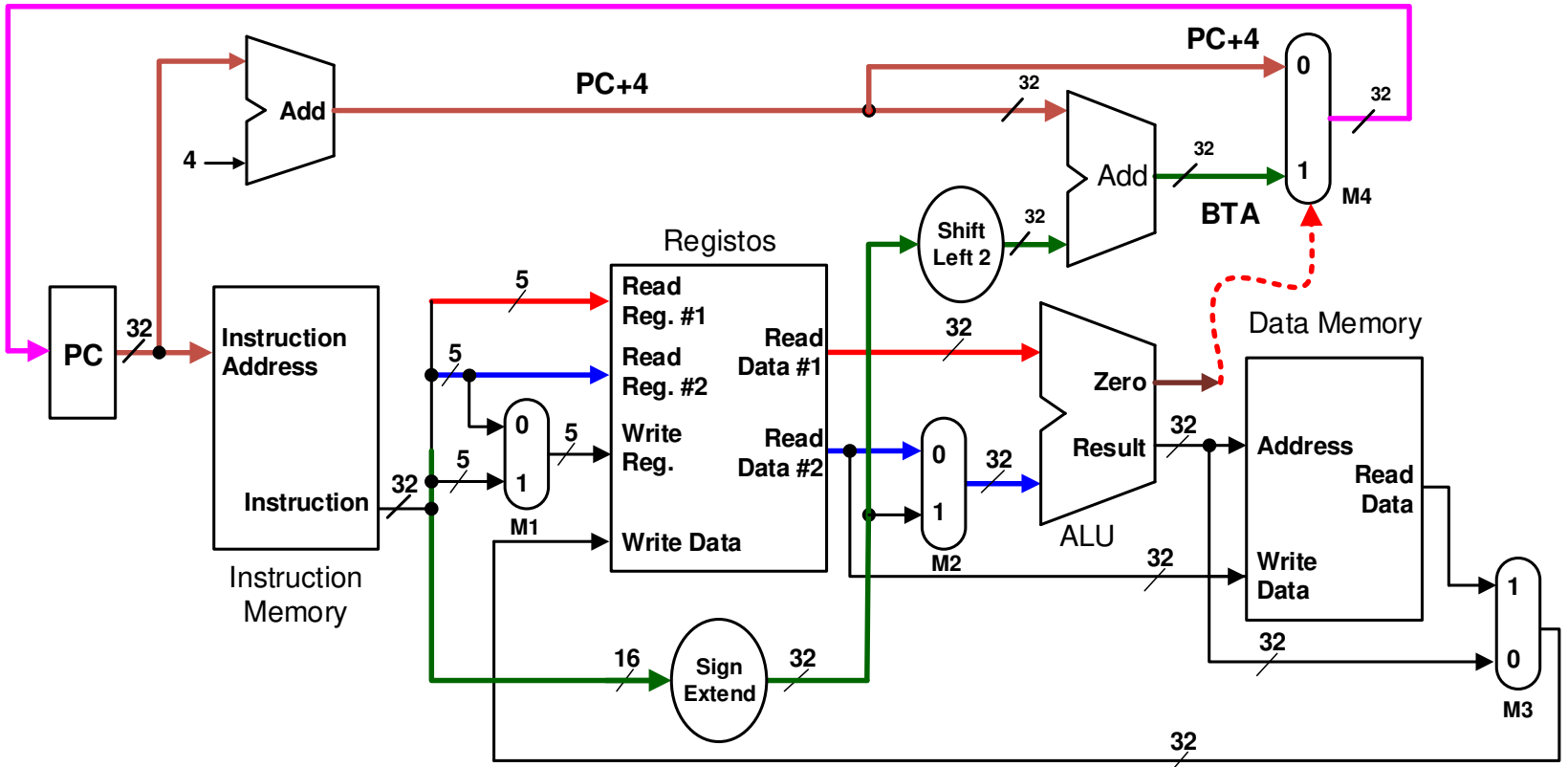
Implementação de um *Datapath* – juntando tudo

- Encaminhamento de PC+4 para a entrada do Program Counter



Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução de *branch* (**beq**)

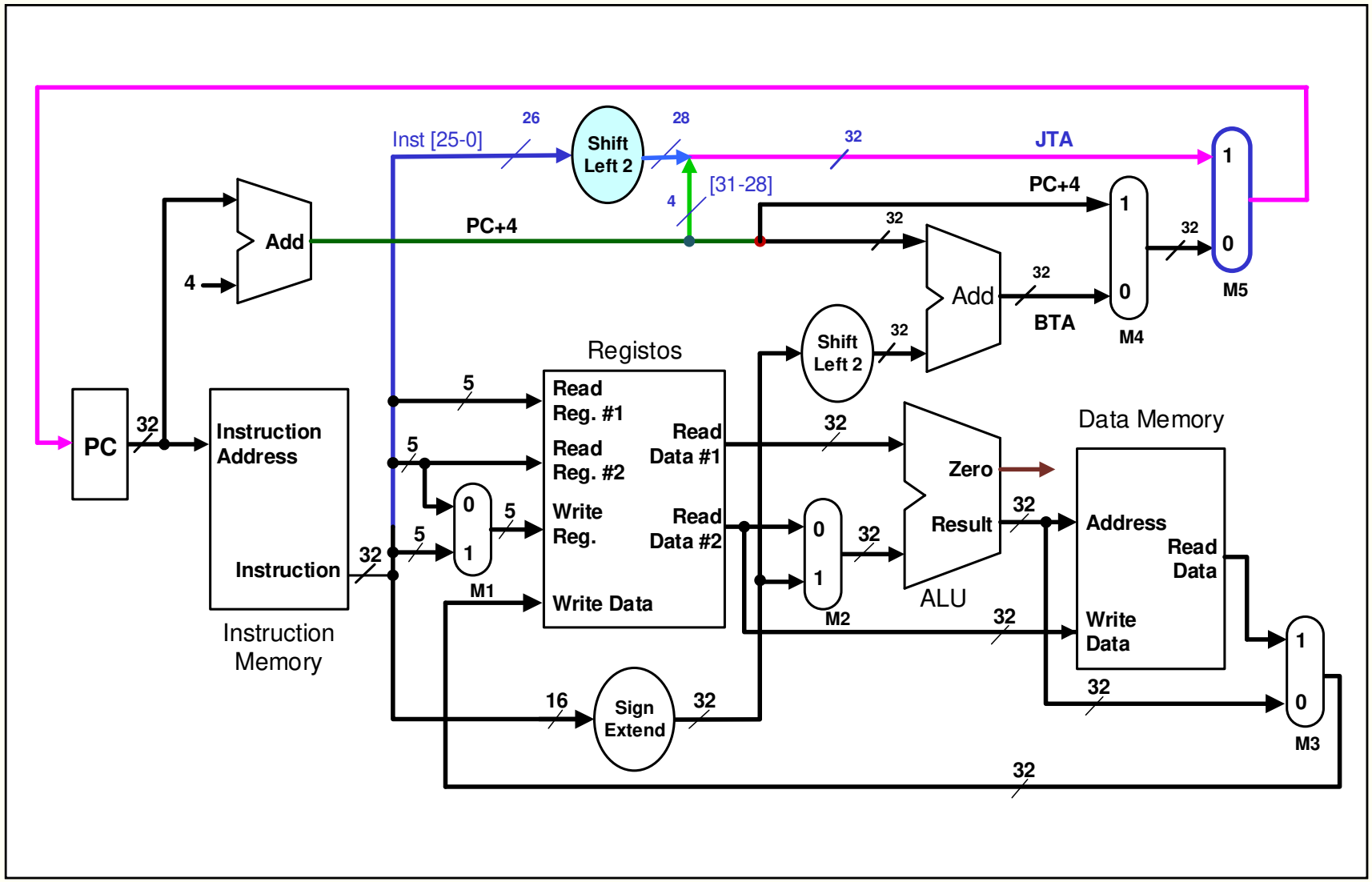


O valor a ser escrito no registo PC, no próximo flanco ativo do relógio, **depende** da saída "zero" da ALU: **"PC+4" se zero=0**; **BTA se zero=1**

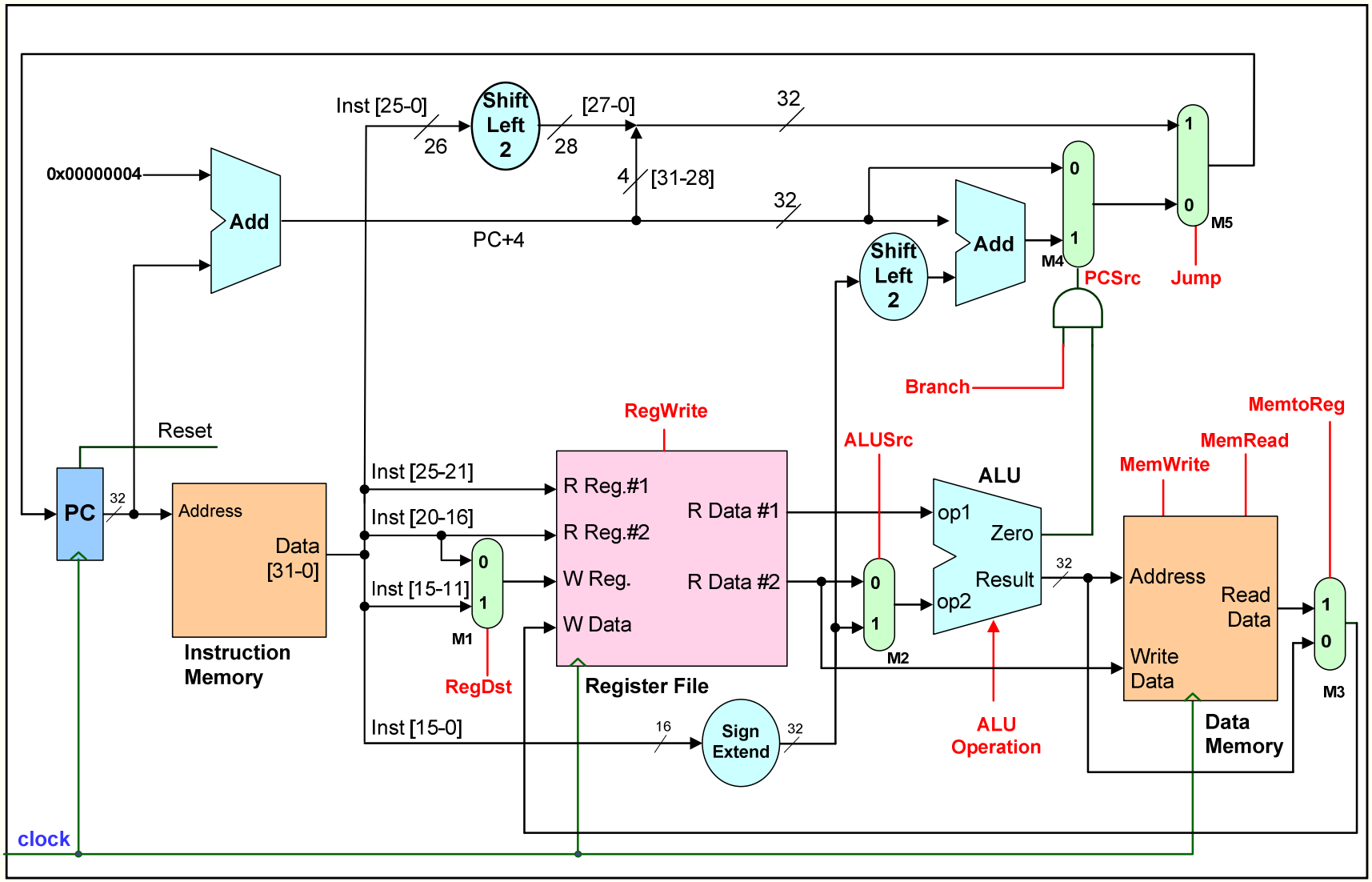
Datapath com suporte para a instrução "j" (jump)

- A instrução “j” é codificada com um caso particular de codificação, o formato J
- No formato J existem apenas dois campos:
  - o campo opcode (**bits 31-26**) e o
  - campo de endereço (**bits 25-0**)
- Na instrução “j”, o endereço alvo (*Jump Target Address* - **JTA**) obtém-se pela **concatenação:**
  - dos bits **31-28** do PC+4 com
  - os bits do campo de endereço da instrução (26 bits) multiplicados por 4 (2 *shifts* à esquerda)
- No próximo flanco ativo do relógio, o valor do PC será **incondicionalmente** alterado com o valor do JTA

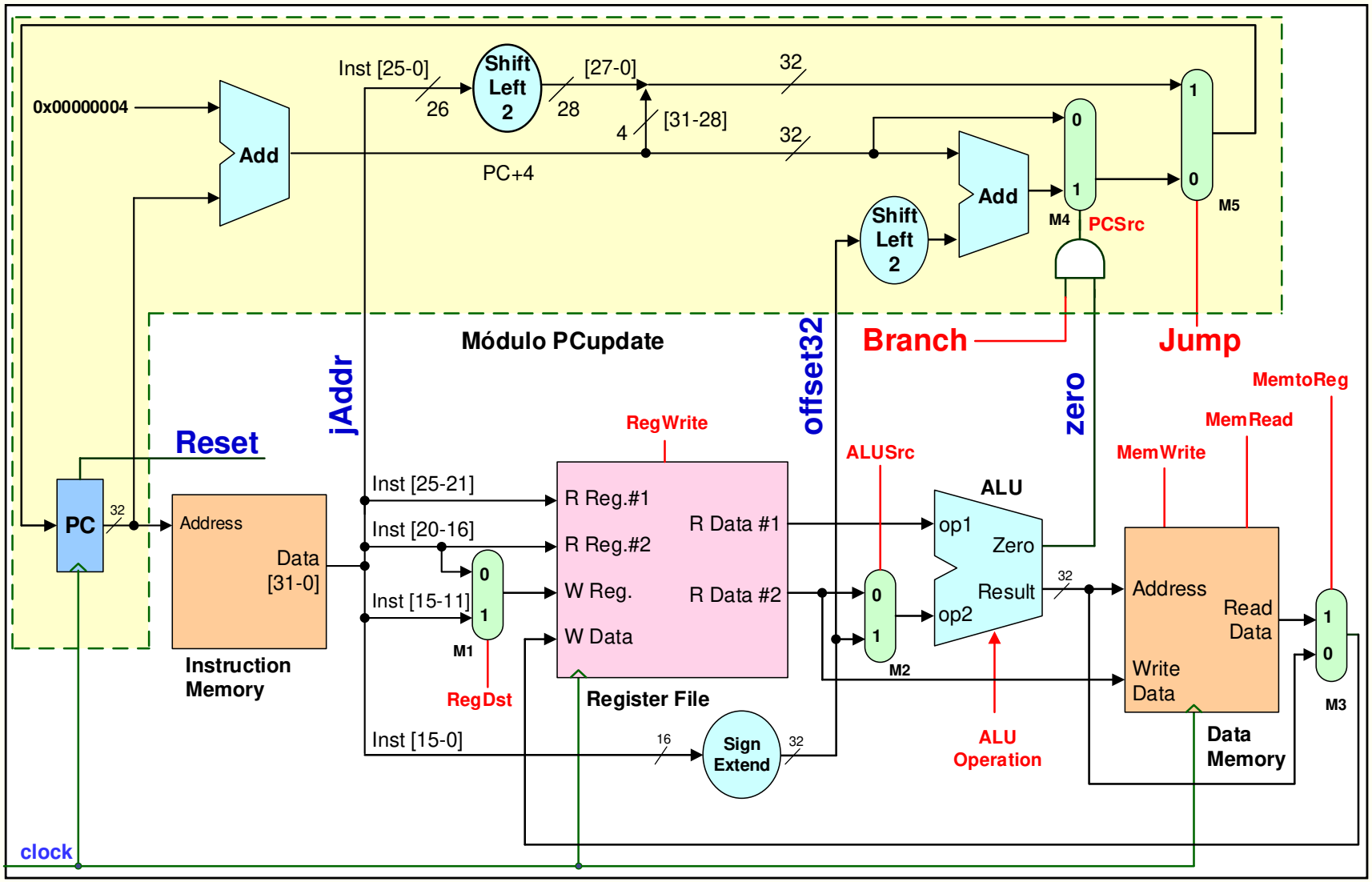
Datapath com suporte para a instrução "jump" (j)



Datapath single-cycle completo (com sinais de controlo)



Módulo de atualização do PC para o DP completo



Módulo de atualização do PC para o DP completo

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

entity PCupdate is
 port(clk       : in std_logic;
      reset     : in std_logic;
      branch    : in std_logic;
      jump      : in std_logic;
      zero      : in std_logic;
      offset32  : in std_logic_vector(31 downto 0);
      jAddr     : in std_logic_vector(25 downto 0);
      pc        : out std_logic_vector(31 downto 0));
end PCupdate;
```

Módulo de atualização do PC para o DP completo

```
architecture Behavioral of PCupdate is
  signal s_pc, s_pc4, s_offset32 : unsigned(31 downto 0);
begin
  s_offset32 <= unsigned(offset32(29 downto 0)) & "00"; -- Left shift
  s_pc4 <= s_pc + 4;
  process(clk)
  begin
    if(rising_edge(clk)) then
        if(reset = '1') then
            s_pc <= (others => '0');
        else
            if(jump = '1') then                    -- Jump Target Address
                s_pc <= s_pc4(31 downto 28) & unsigned(jAddr) & "00";
            elsif(branch = '1' and zero = '1') then
                s_pc <= s_pc4 + s_offset32;      -- Branch Target Address
            else
                s_pc <= s_pc4;
            end if;
        end if;
    end if;
  end process;
  pc <= std_logic_vector(s_pc);
end Behavioral;
```

S

Exercícios

- Quais as diferenças entre uma arquitetura Harvard e uma arquitetura von Neumann?
- Suponha um sistema baseado numa arquitetura von Neumann, com um barramento de endereços de 20 bits e com uma organização de memória do tipo *byte-addressable*. Qual a dimensão máxima, em bytes, que os programas a executar neste sistema (instruções+dados+stack) podem ter?
- Num processador baseado numa arquitetura Harvard, a memória de instruções está organizada em *words* de 32 bits, a memória de dados em words de 8 bits (*byte-addressable*) e os barramentos de endereços respetivos têm uma dimensão de 24 bits. Qual a dimensão, em bytes, dos espaços de endereçamento de instruções e de dados?
- O que significa um elemento de estado ter escrita síncrona?
- Considere um elemento de estado, com leitura assíncrona, que apenas tem o sinal de *clock*, na sua interface de controlo. O que pode concluir-se relativamente à escrita?

Exercícios

- Suponha um elemento de estado, com escrita síncrona e leitura assíncrona, que apresenta, na sua interface de controlo, um sinal "read", um sinal "write" e um sinal de *clock*. Indique que sinal ou sinais têm que estar ativos para que se realize: a) uma operação de leitura; b) uma operação de escrita.
- Qual a capacidade de armazenamento, expressa em bytes, de uma memória com uma organização interna em *words* de 32 bits e um barramento de endereços de 30 bits?
- Quais as operações realizadas no *datapath* que são comuns a todas as instruções?
- Identifique a operação realizada na ALU na realização de cada uma das seguintes instruções: tipo R, addi, slti, lw, sw e beq.
- Indique qual a operação realizada na conclusão de cada uma das seguintes instruções: tipo R, addi, slti, lw, sw, beq e j.
- Suponha que o *datapath* está a executar a instrução **`add $3,$4,$5`**. Que operações serão realizadas na próxima transição ativa do sinal de relógio?
- No *datapath single-cycle* que tipo de informação é armazenada na memória cujo endereço é a saída do registo PC?

Exercícios

- Qual o endereço de memória onde deve estar armazenada a primeira instrução do programa para que a execução possa ser reiniciada sempre que se ative o sinal de "reset" do registo PC?
- Suponha que cada registo do banco de registos foi inicializado com um valor igual a: (32-número do registo). Indique o valor presente nas entradas do banco de registos **`ReadReg1`**, **`ReadReg2`** e **`WriteReg`**, e o valor presente nas saídas **`ReadData1`** e **`ReadData2`**, durante a execução das instruções com o código máquina: **`0x00CA9820`**, **`0x8D260018`** (**`lw`**) e **`0xAC6A003C`** (**`sw`**).
- Considerando ainda a inicialização do banco de registos da questão anterior, indique qual o valor calculado pela ALU durante a execução das instruções LW com o código máquina **`0x8CA40005`** e **`0x8CE6FFF3`**.
- Qual o valor à saída do somador de cálculo do BTA durante a execução da instrução cujo código máquina é **`0x10430023`**, supondo que o valor à saída do registo PC é **`0x00400034`**?