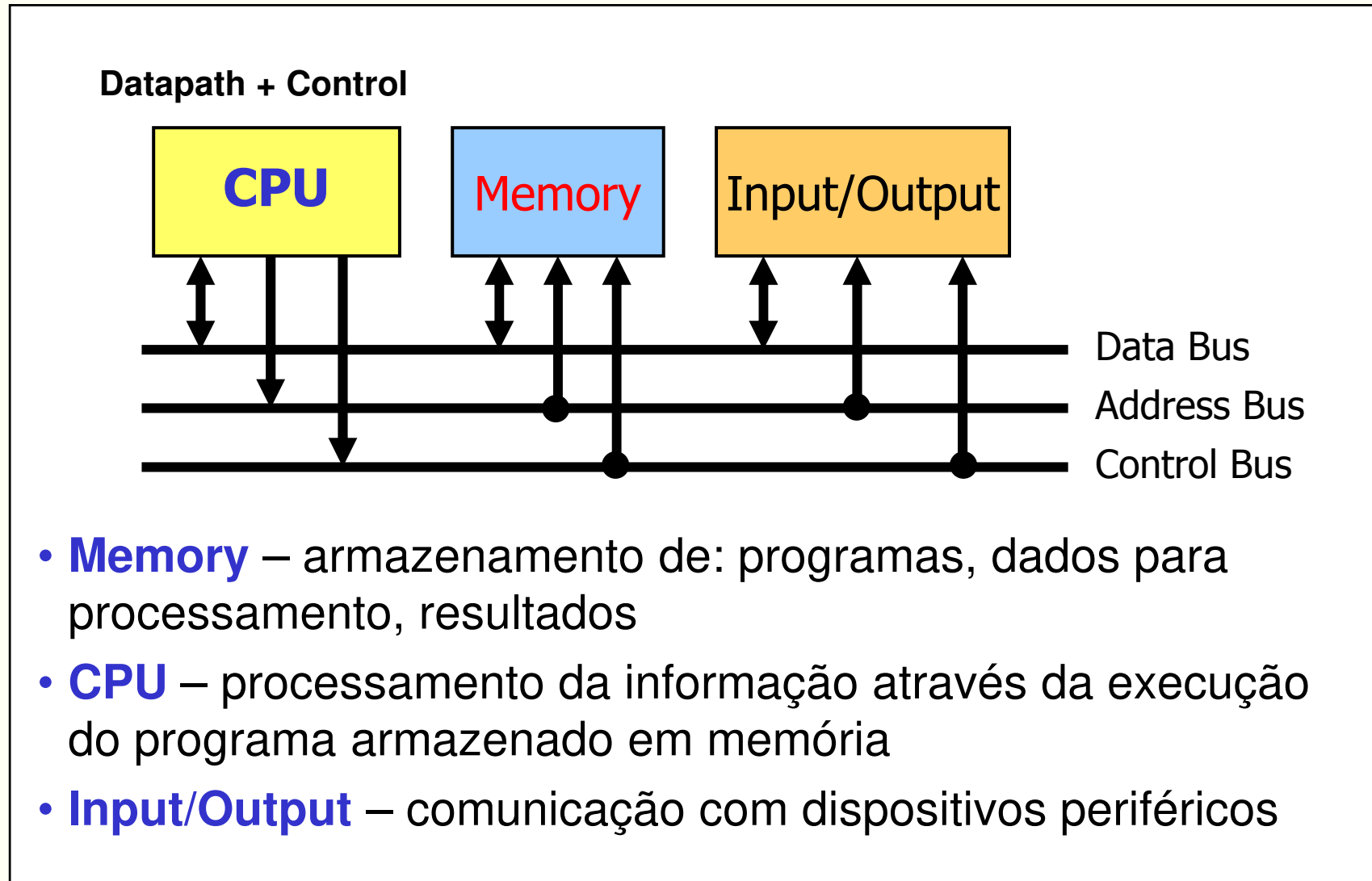


Aulas 13, 14 e 15

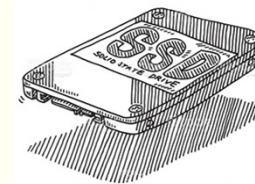
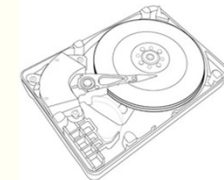
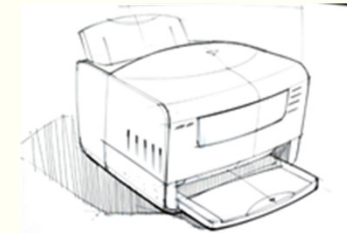
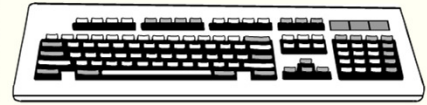
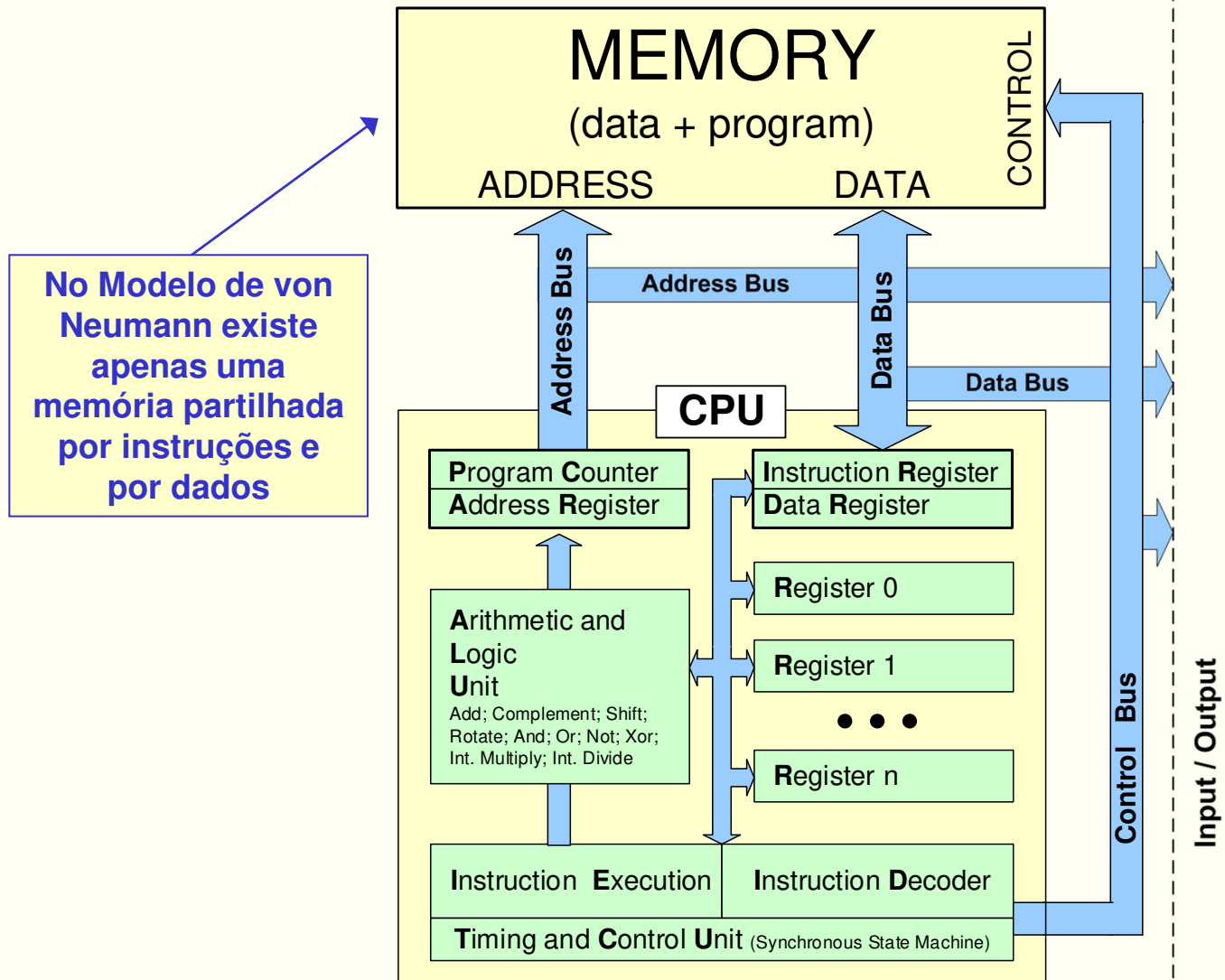
- Modelos de Harvard e Von Neumann
- Pressupostos para a construção de um *datapath* genérico para uma arquitetura tipo MIPS
- Análise dos blocos constituintes necessários à execução de um subconjunto de instruções de cada uma das classe de instruções:
 - Aritméticas e lógicas (add, addi, sub, and, or,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

José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador

Modelo de von Neumann

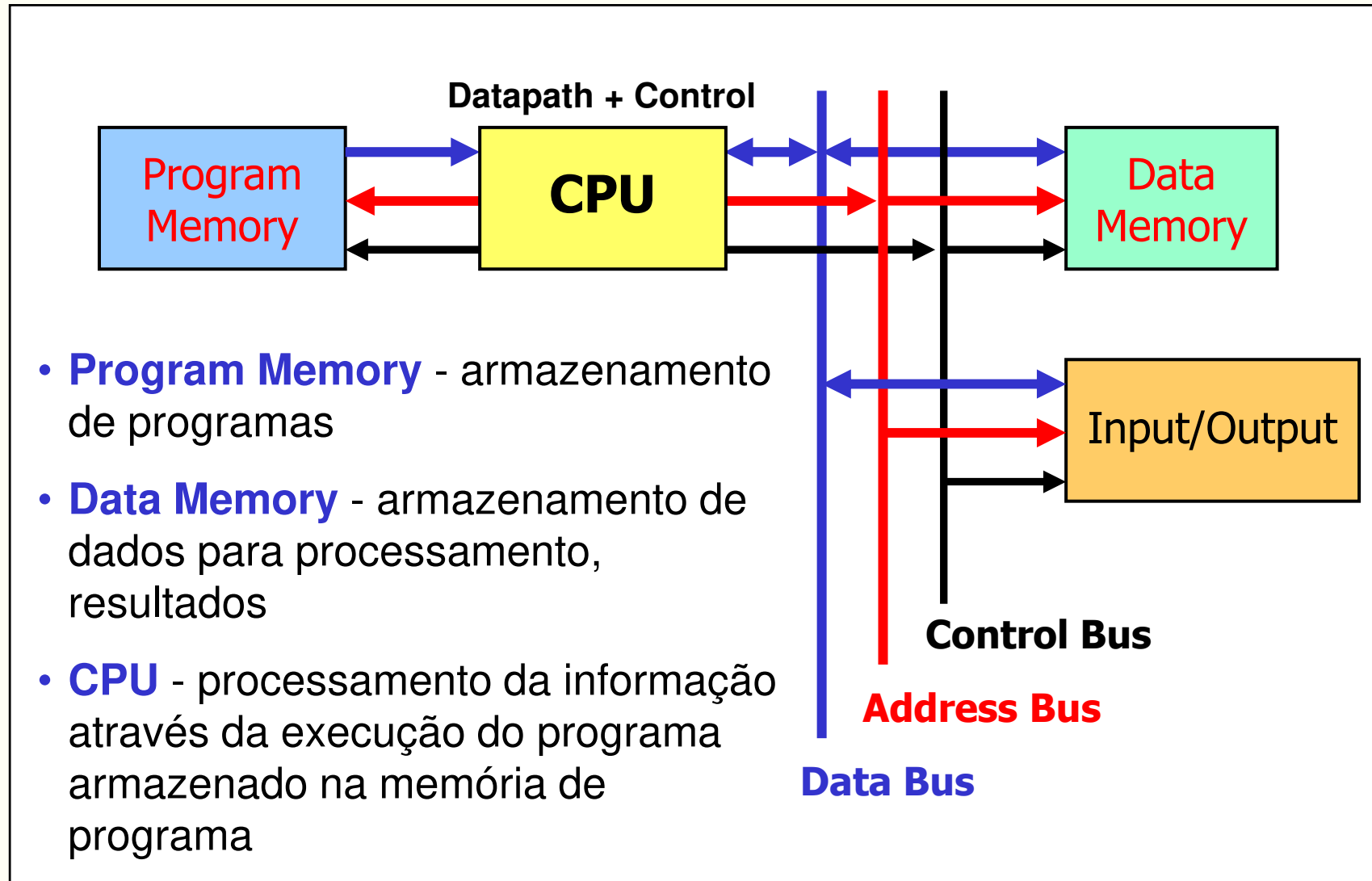


Modelo de von Neumann



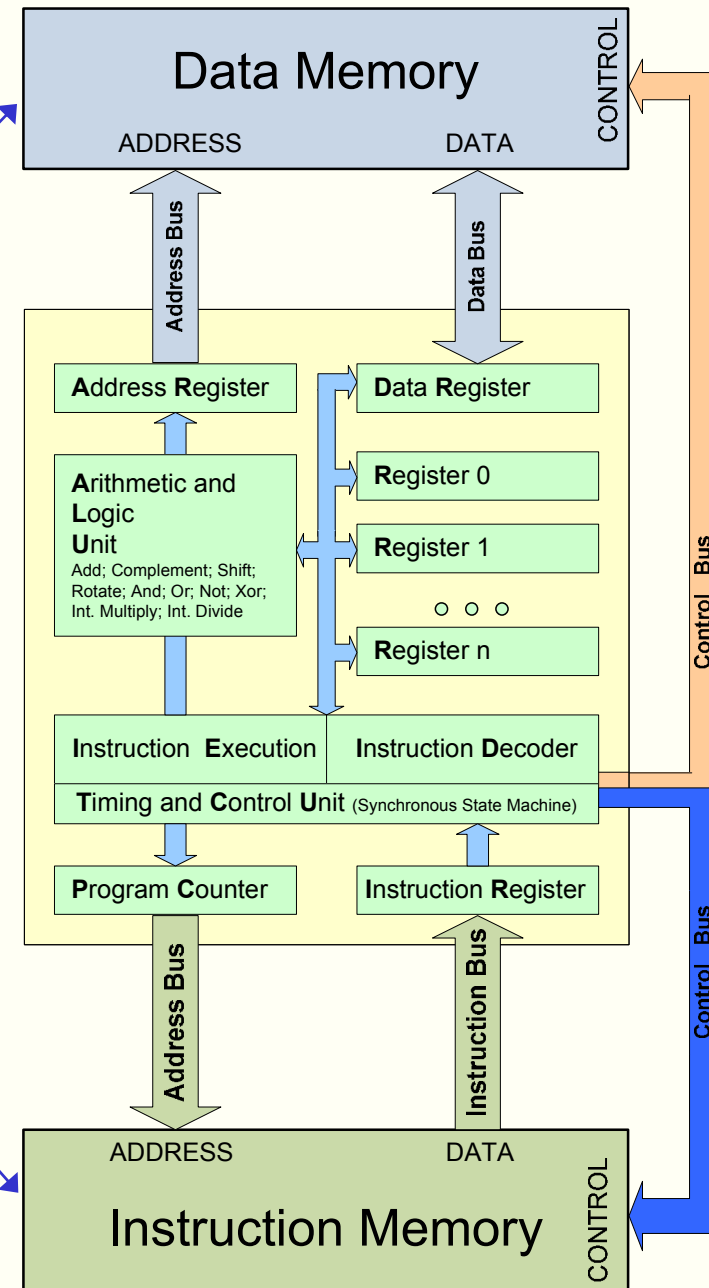
Input / Output

Modelo de Harvard



Modelo de Harvard

No Modelo de Harvard existem duas memórias independentes (uma para instruções e outra para dados) que podem ser acedidas simultaneamente pelo CPU



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 dimensões de palavra diferentes



Implementação de um *Datapath*

- O CPU consiste, fundamentalmente, em duas secções:
 - **Secção de dados** - elementos operativos/funcionais para armazenamento, processamento e encaminhamento da informação:
 - Registos internos
 - 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 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 a seguir considera-se a memória externa ao CPU 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

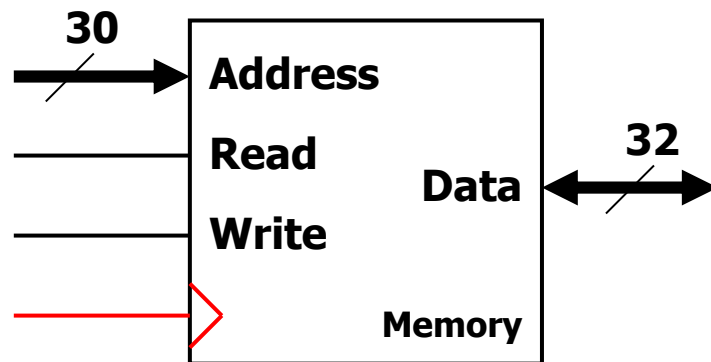
NOTA: Nos slides seguintes, havendo um sinal de relógio comum, e por uma questão de simplificação dos diagramas, o sinal de relógio pode não ser explicitamente representado



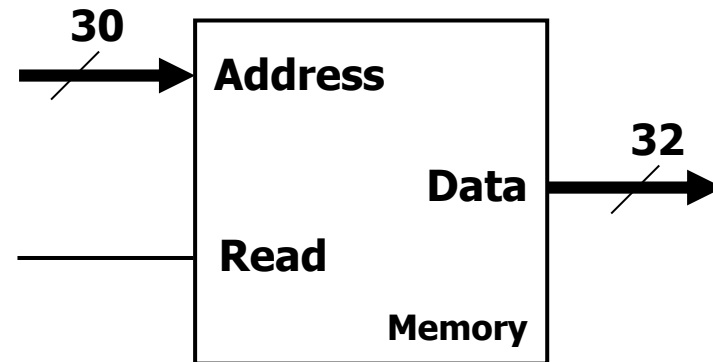
Implementação de um *Datapath*

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

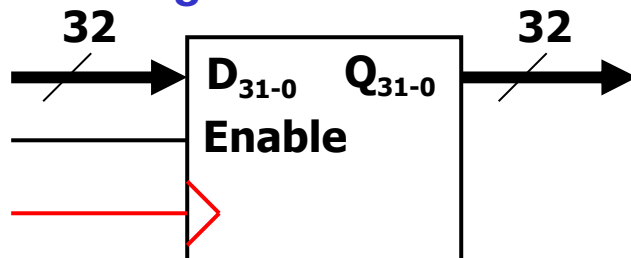
Memória para escrita e leitura (2^{30} words de 32 bits)



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



Registro 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*

- 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, slti**)
 - Instruções de acesso à memória: load word (**lw**) e store word (**sw**)
 - As instruções de salto condicional (**beq, bne**) e salto incondicional (**j**)
- Como se verá, 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*

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



Implementação de um *Datapath*

- 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 classes de instruções** (à exceção do salto incondicional) **utilizam a ALU depois da leitura dos registos**:
 - as instruções aritméticas e lógicas para a execução da 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) – o novo valor é obtido a partir dos 26 LSB do código máquina da instrução e dos 4 bits mais significativos do valor atual do PC (ver aula 6)



Implementação de um *Datapath*

- 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 de **branch** 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 testada ser verdadeira



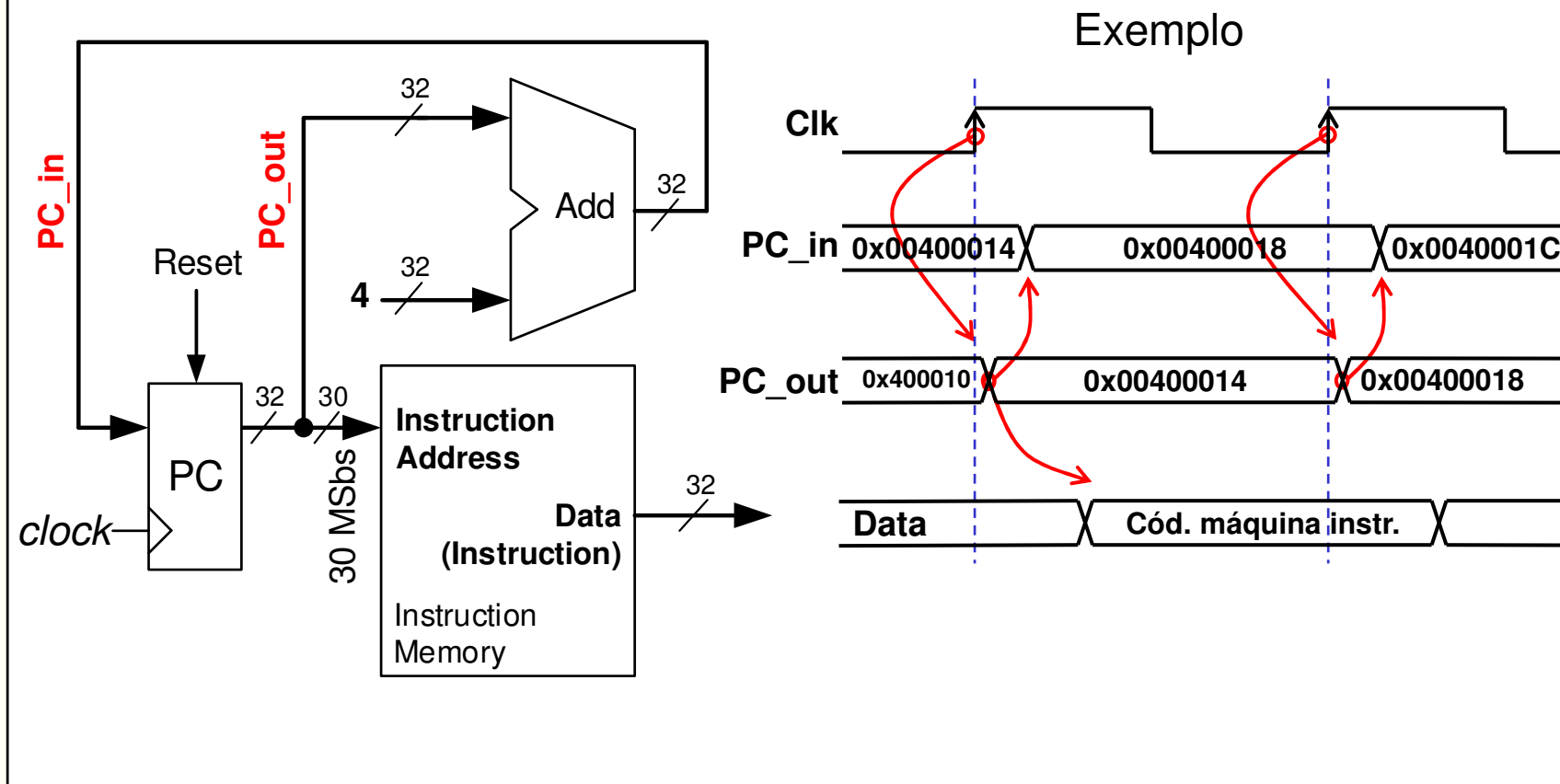
Implementação de um *Datapath* – *Instruction Fetch*

- O processo de acesso à memória para leitura da próxima instrução é genericamente designado por ***Instruction Fetch***
- Por uma questão de simplificar a organização da informação, 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*

- 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**)
 - Instruções de salto condicional (Tipo I: **beq**, **bne**)

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 na execução de uma instrução do 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

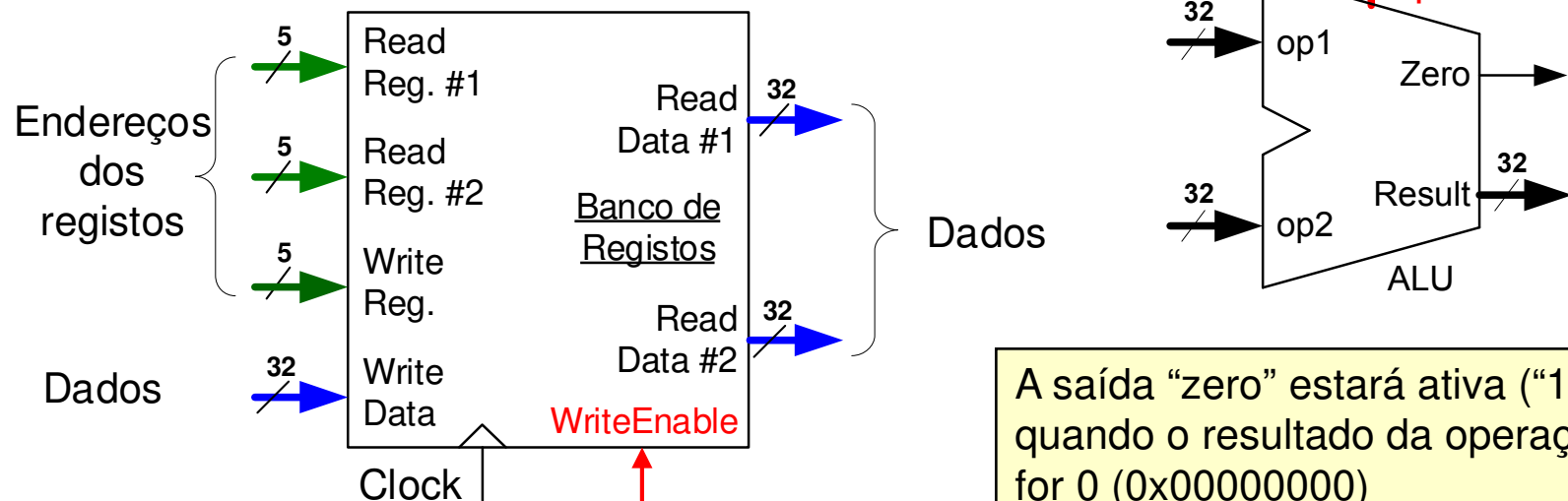


Código máquina: 0x00641020



Implementação de um *Datapath* – instruções tipo R

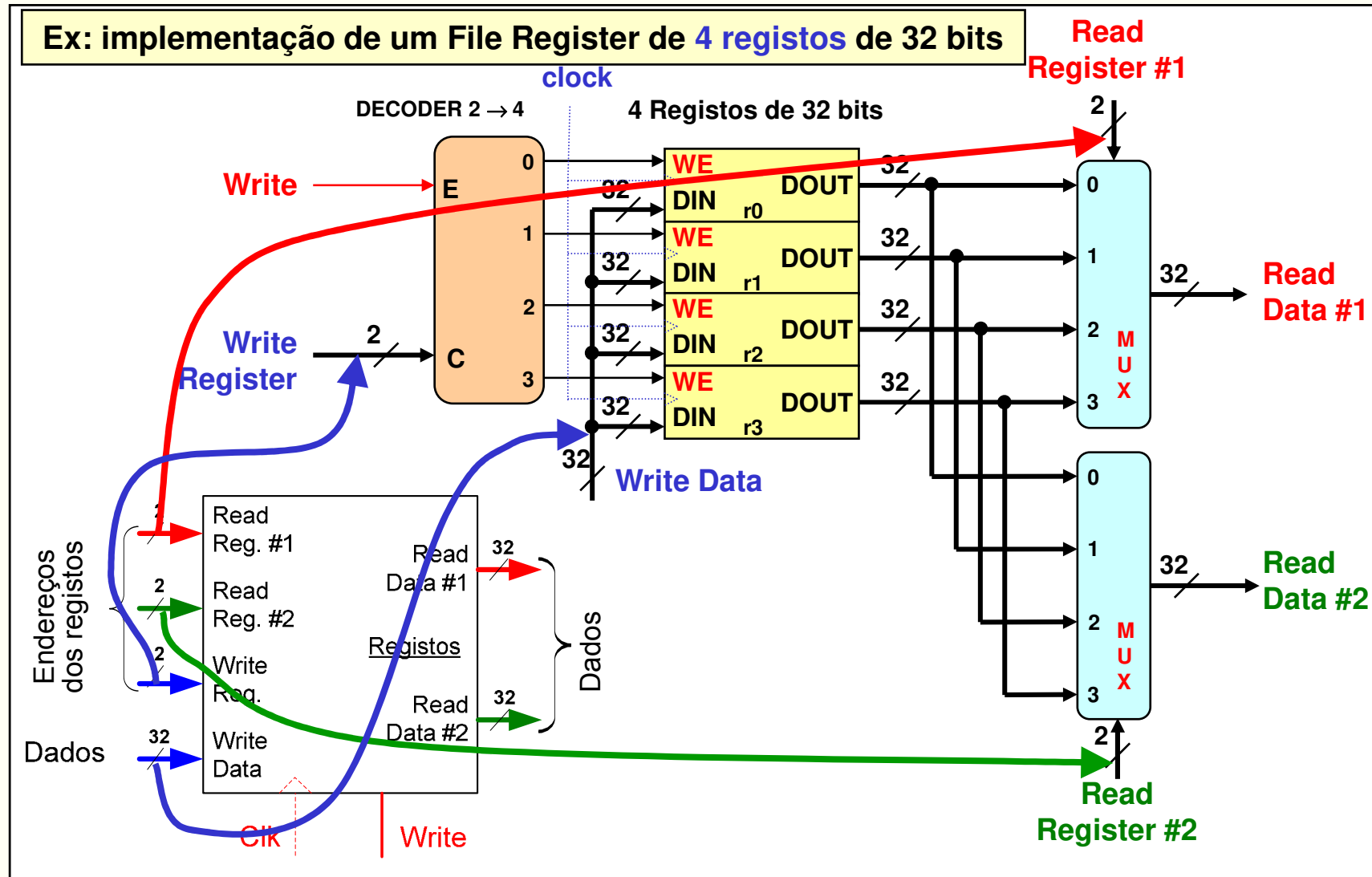
- Os elementos necessários à execução das instruções aritméticas e lógicas (tipo R) são:
 - Uma ALU de 32 bits
 - Um conjunto de registros internos (Banco de registros com 32 registros de 32 bits cada)



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



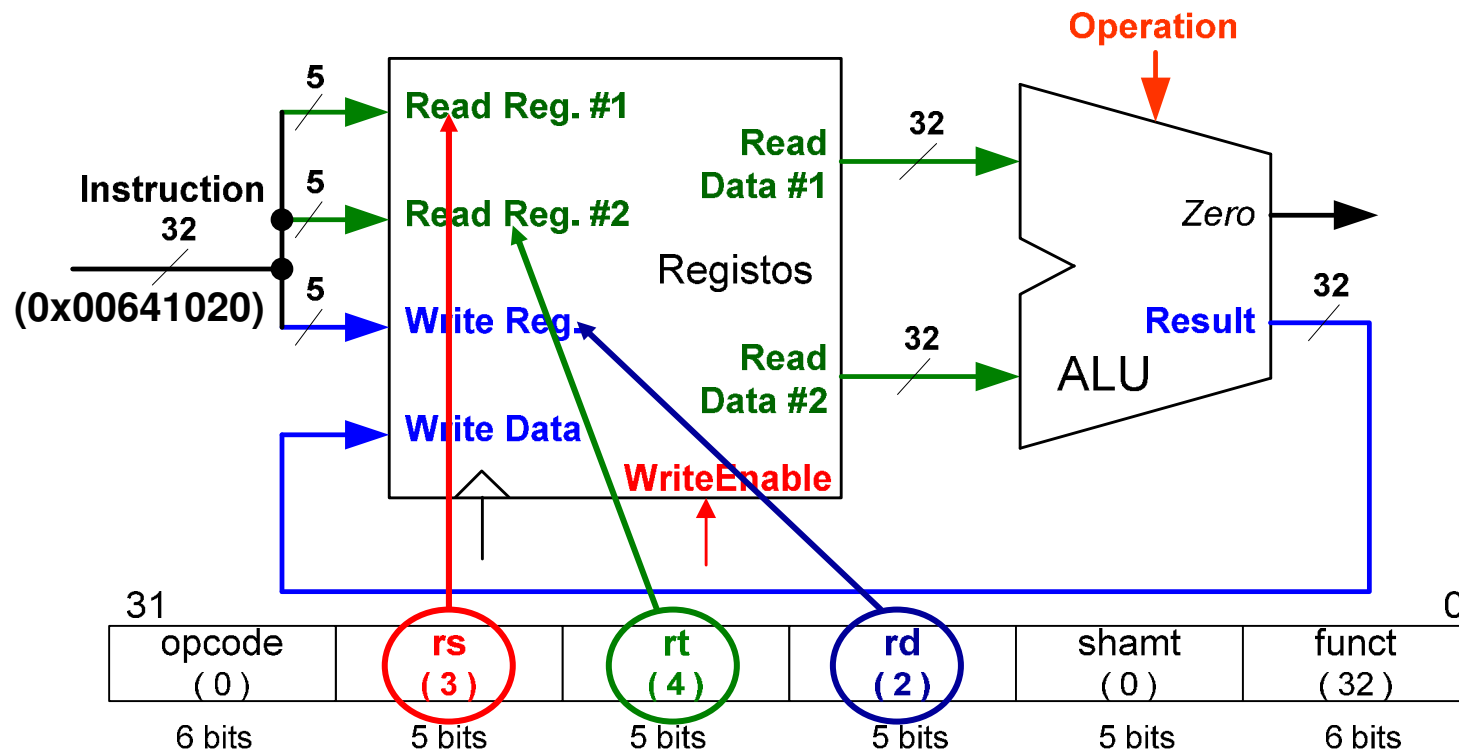
Implementação de um *Datapath* – instruções tipo R



Implementação de um *Datapath* – instruções tipo R

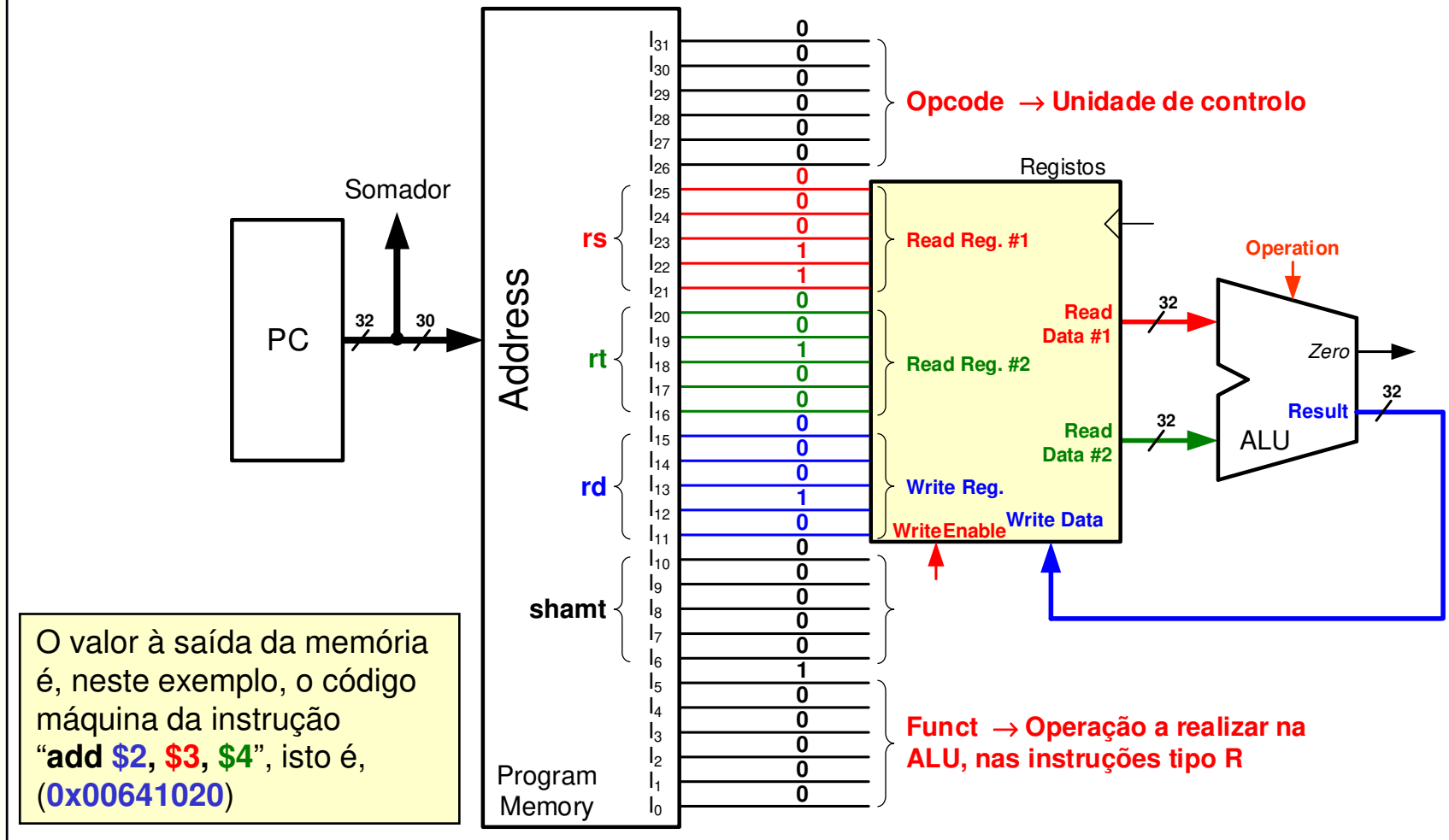
- A interligação dos elementos operativos será:

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



Implementação de um *Datapath* – instruções tipo R

- Ligação entre a memória de código e o Banco de Registos (Instruções tipo R)



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

opcode (43)	rs (4)	rt (2)	offset (0x24)
------------------	--------------------	--------------------	---------------------------



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)

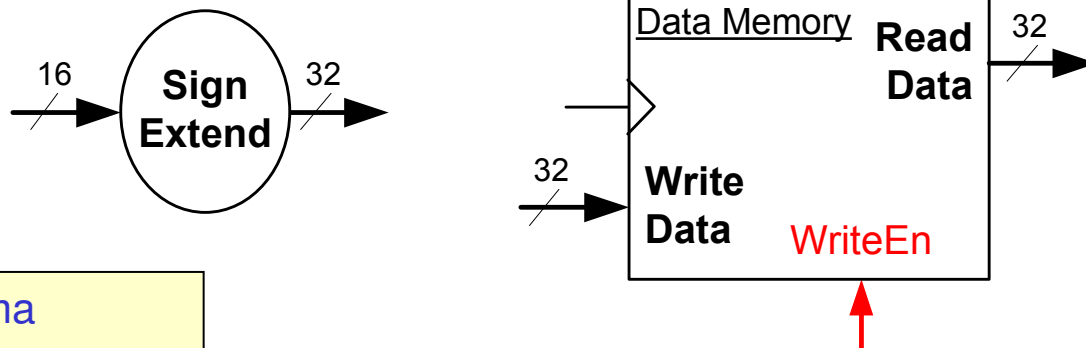
Endereço inicial da memória para leitura de uma word de 32 bits (vai ser escrita no registo \$4)

opcode (35)	rs (15)	rt (4)	offset (0x2F)
------------------	--------------	-------------	--------------------



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

- Os elementos necessários à execução das instruções de transferência de informação entre registros 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

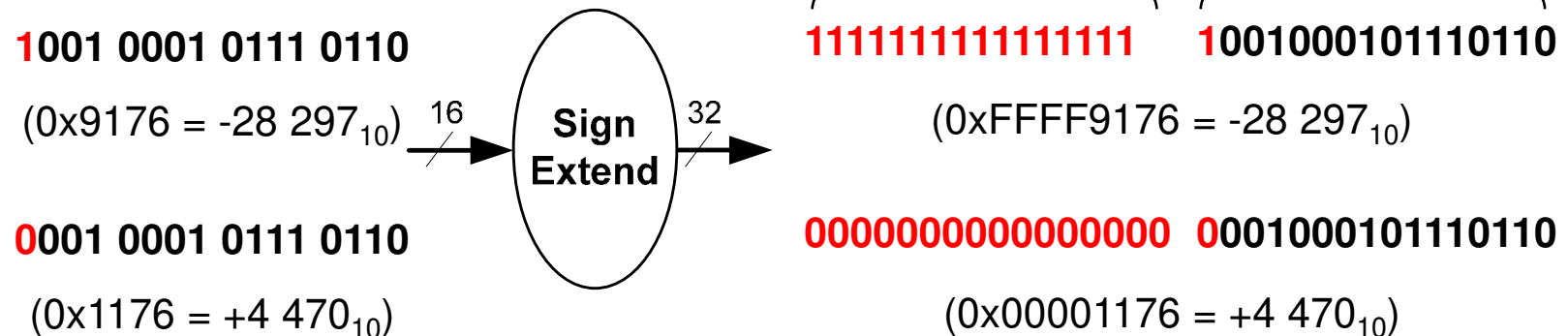


Implementação de um *Datapath* (Instruções lw e sw)

Módulo de extensão de sinal

- 16 bits são apresentados na entrada
- 32 bits são disponibilizados na saída

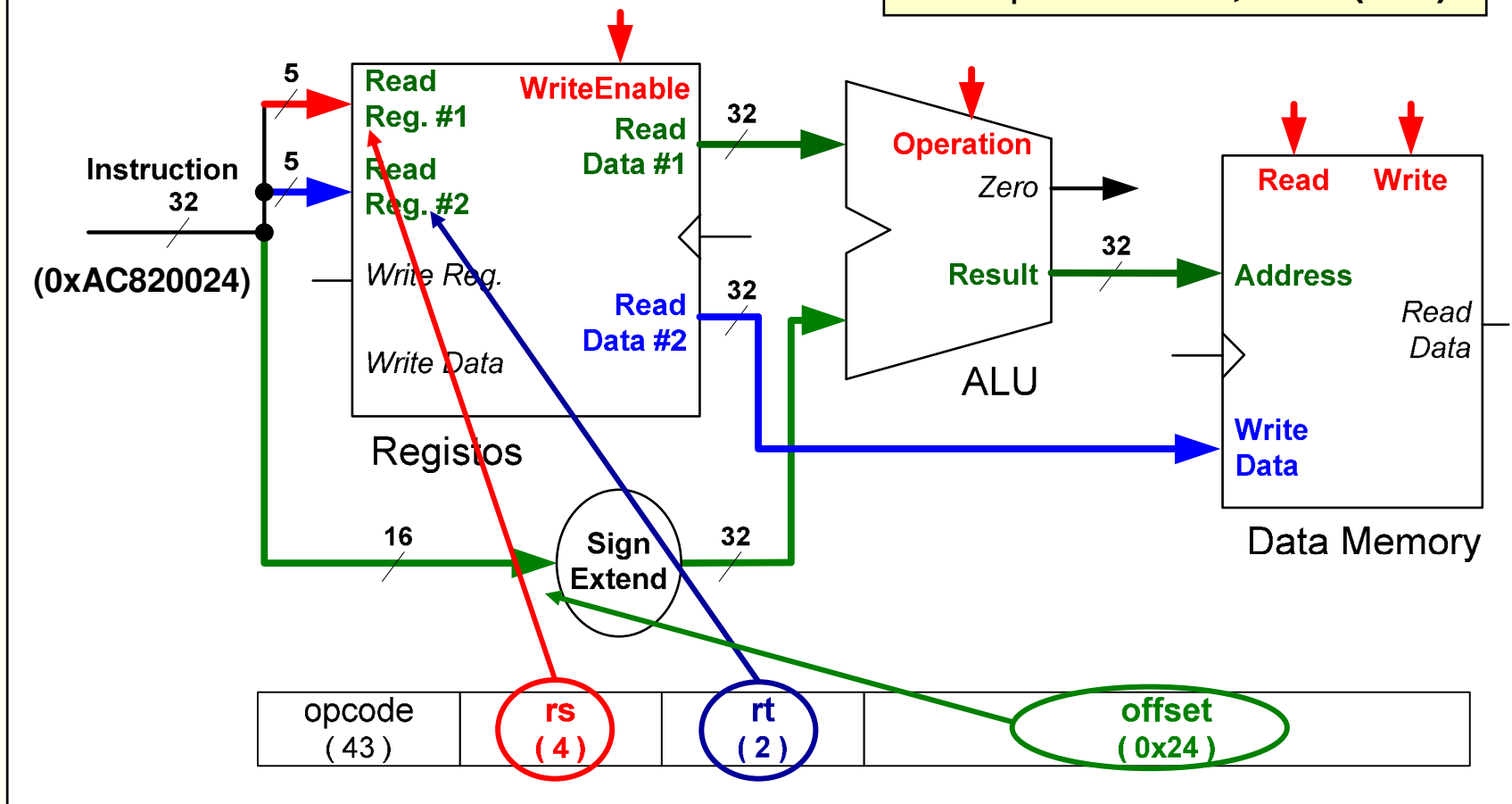
Exemplos:



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

- A interligação dos elementos operativos necessários à execução do “**sw**” será:

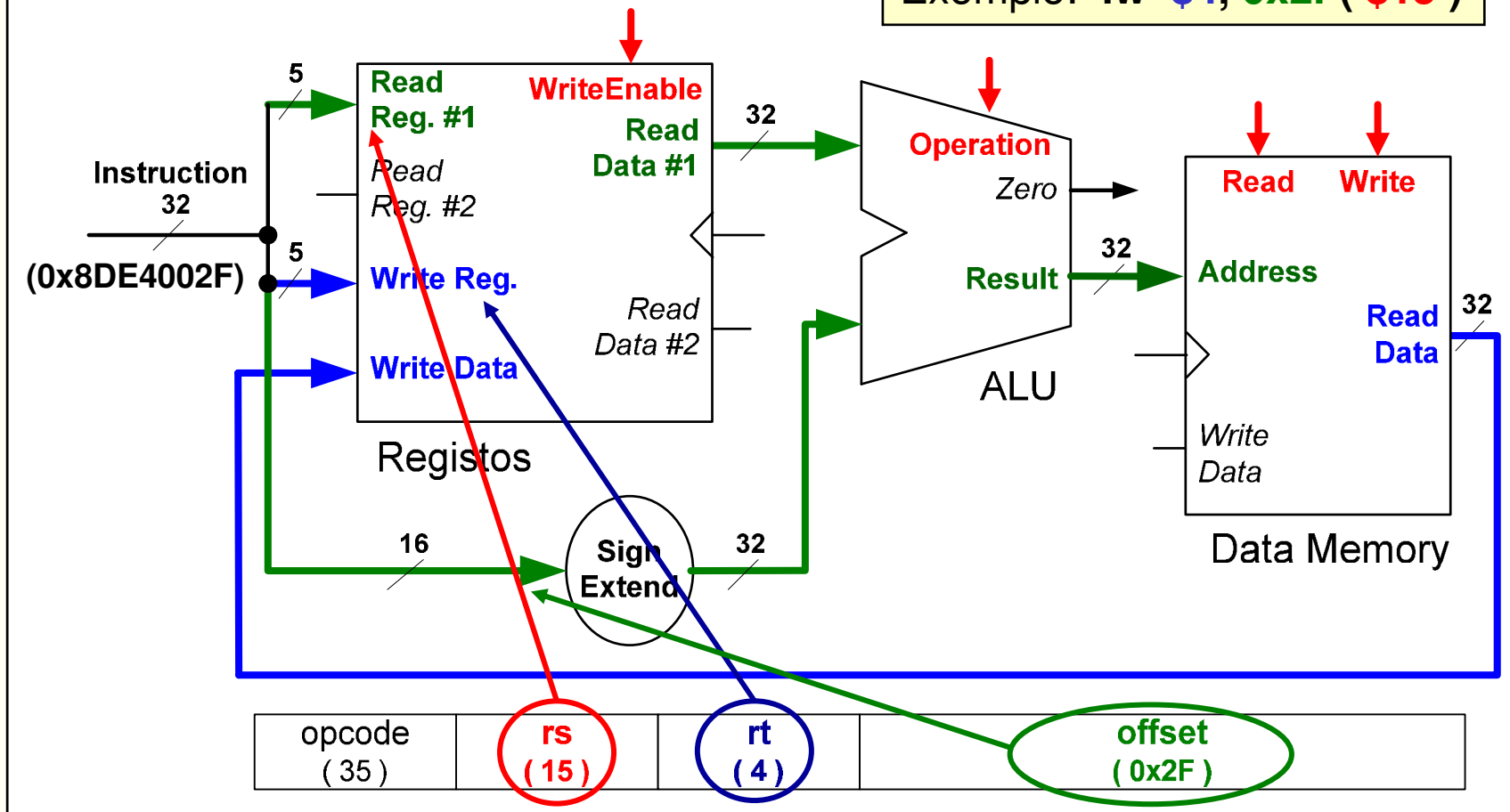
Exemplo: **sw** \$2, 0x24(\$4)



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

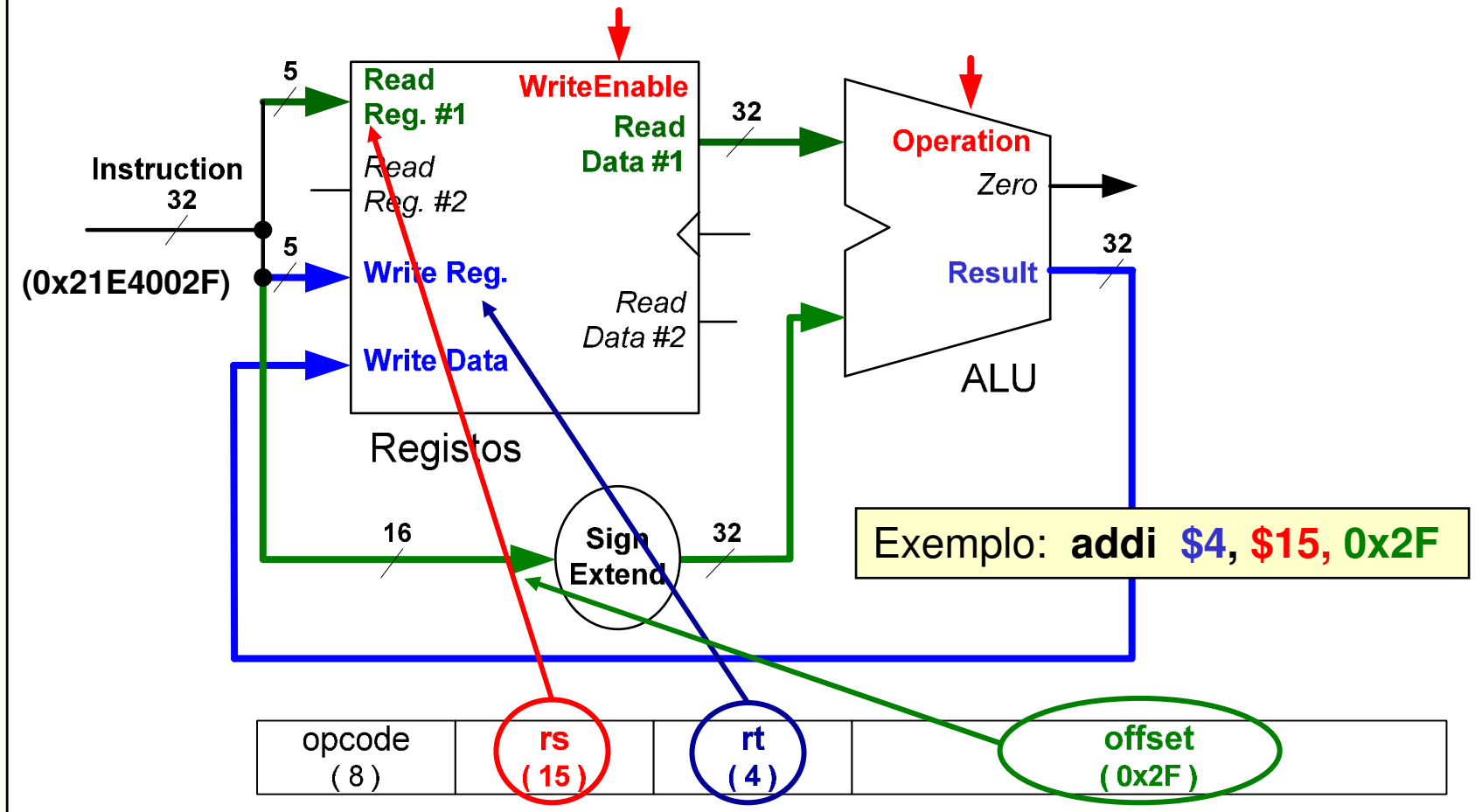
- A interligação dos elementos operativos necessários à execução do “lw” será:

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



Implementação de um *Datapath* (Instruções “imediatas”)

- A interligação dos elementos operativos necessários à execução de instruções que operam com constantes (“**addi**”, “**slti**”) será:



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 dos registos a comparar
 - Comparação dos valores dos registos (realização de uma operação de subtração na ALU)
 - Cálculo do endereço-alvo da instrução de *branch*
(*Branch Target Address* - BTA) - ver aula 6
$$\text{BTA} = (\text{PC} + 4) + (\text{instruction_offset} * 4)$$
 - 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

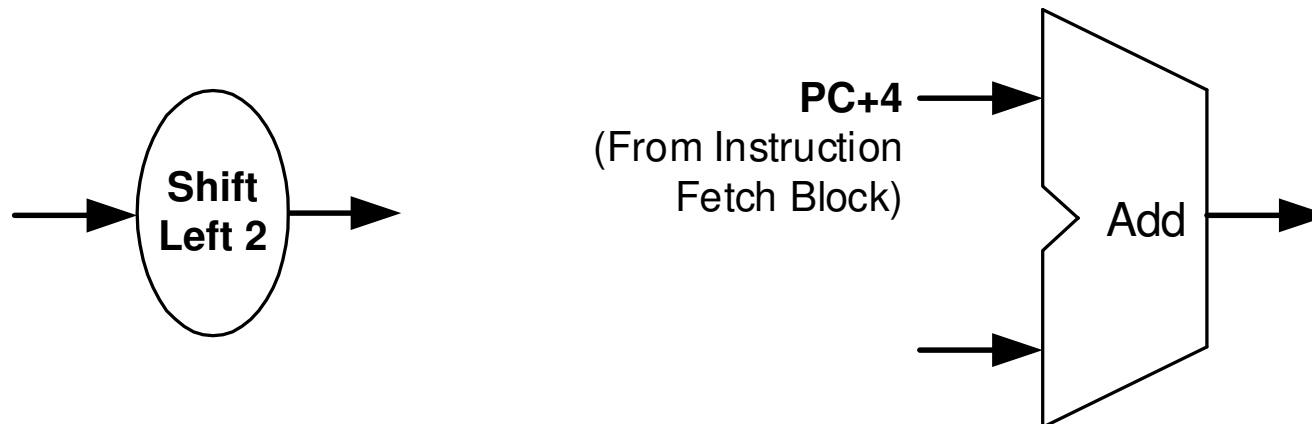
Exemplo: **beq** \$2, \$3, 0x20

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



Implementação de um *Datapath* (Instruções de branch)

- Finalmente, 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 do *instruction offset* que são desprezados no momento da codificação da instrução (ver aula 6)

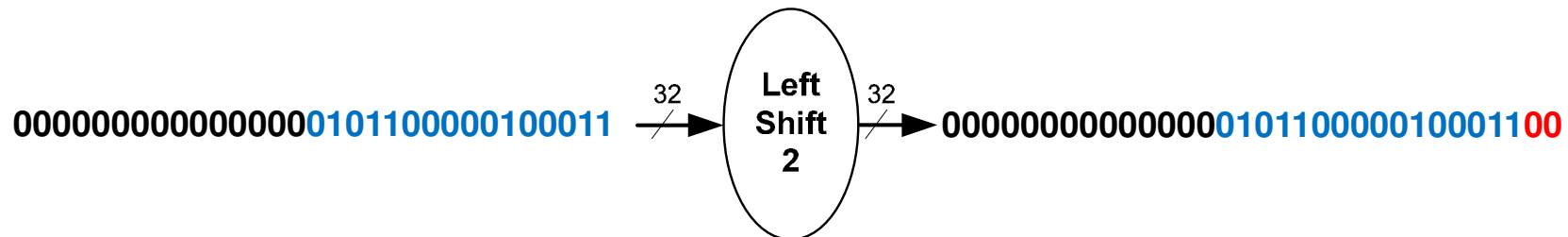


Implementação de um *Datapath* (Instruções lw e sw)

Módulo de left shift

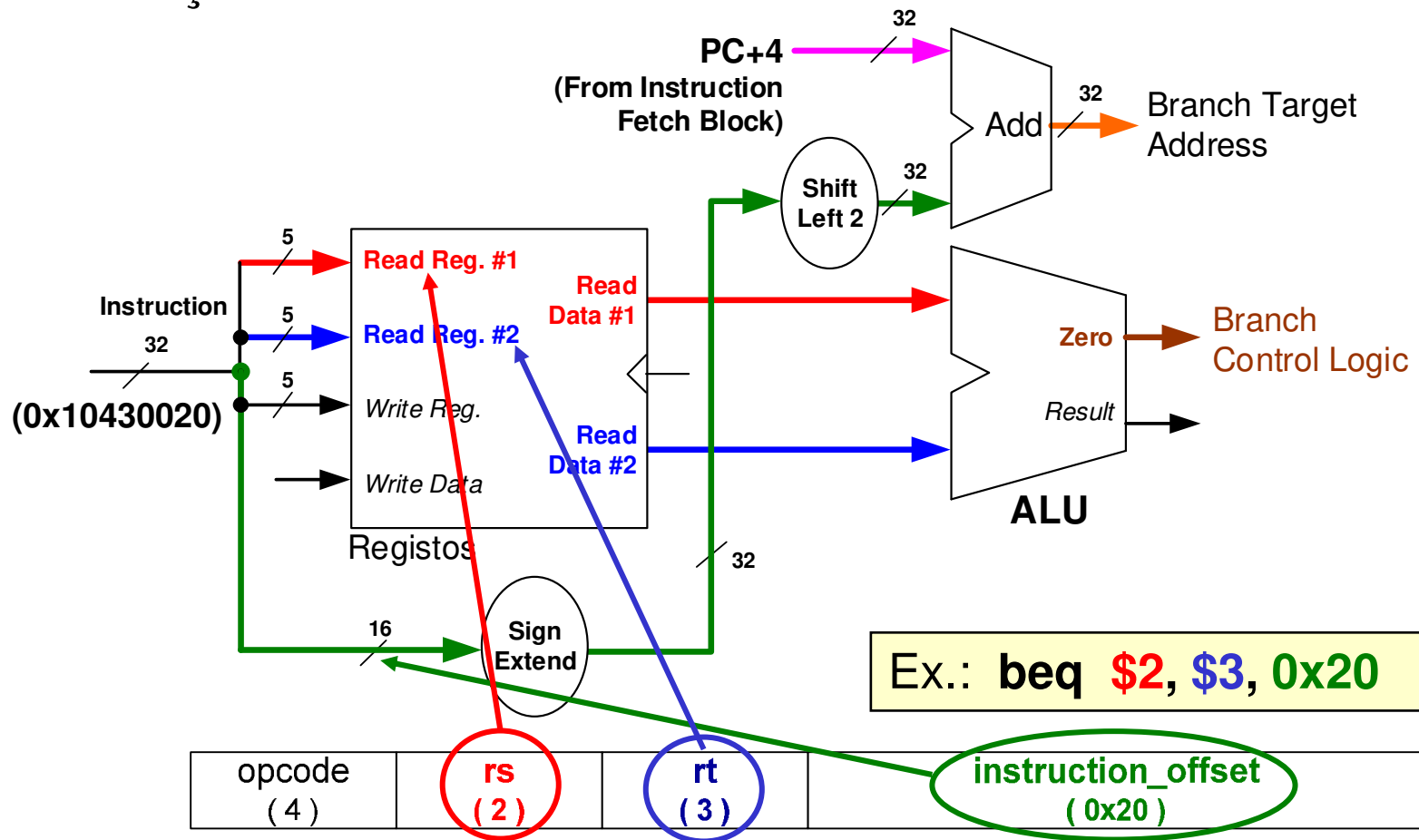
- 32 bits são apresentados na entrada
- Saída = 32 bits à entrada * 4

Exemplo:



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

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



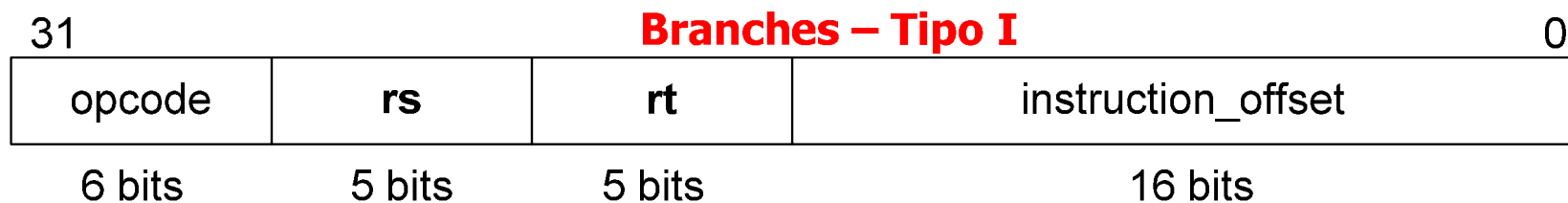
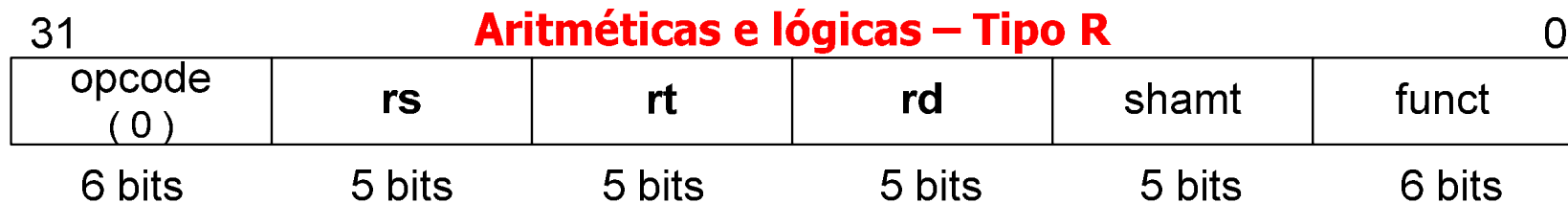
Implementação de um *Datapath* – juntando tudo

- Nos slides anteriores identificaram-se, separadamente, os blocos básicos constituintes do *Datapath* necessários à execução dos vários tipos de instruções
- **Como juntar e interligar os diversos blocos, por forma a servir todas as instruções?**
 - Identificação dos blocos que podem ser partilhados pelos vários tipos de instruções
 - Desenvolvimento de uma estratégia que permita que os mesmos possam ser “configurados” para cada caso
- (o suporte para a instrução J (jump) será introduzido mais tarde)



Implementação de um *Datapath* – juntando tudo

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

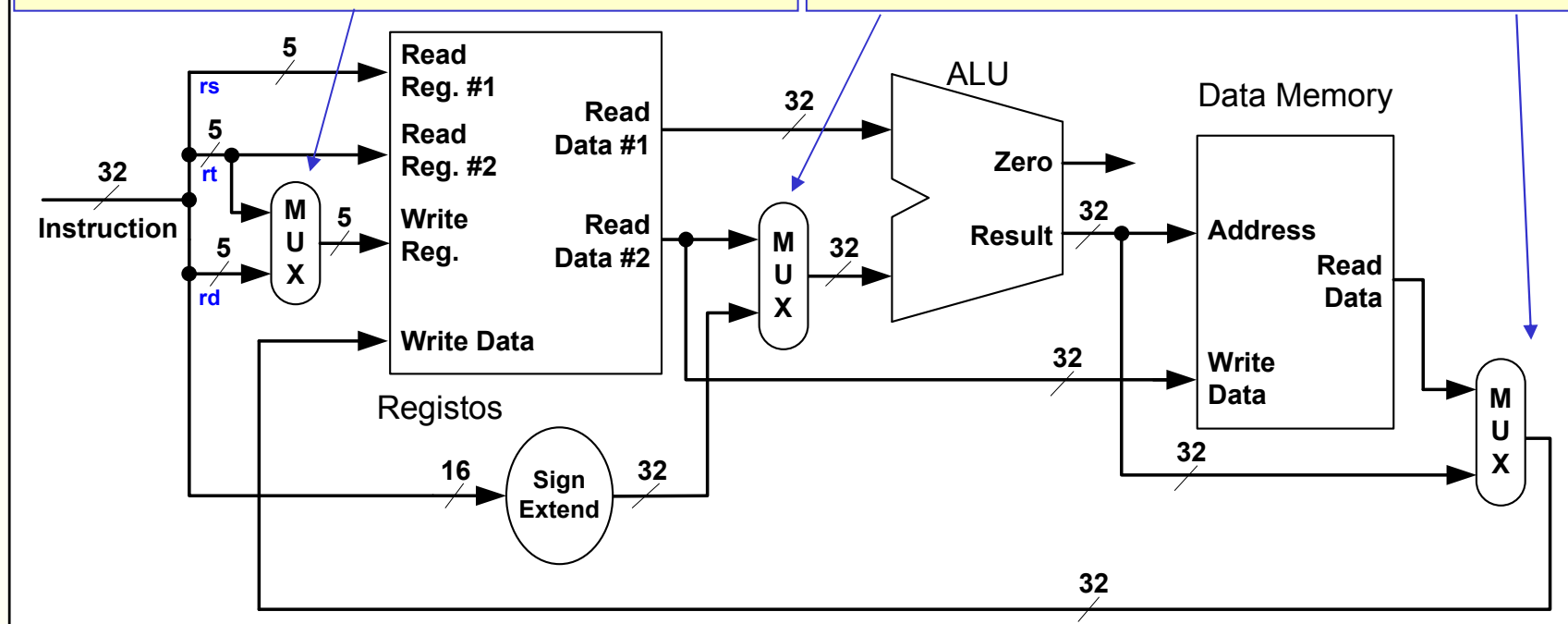


Implementação de um *Datapath* – juntando tudo

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

Escolha do registo destino (3º campo nas instruções tipo R, 2º na instrução LW e nas aritméticas e lógicas imediatas)

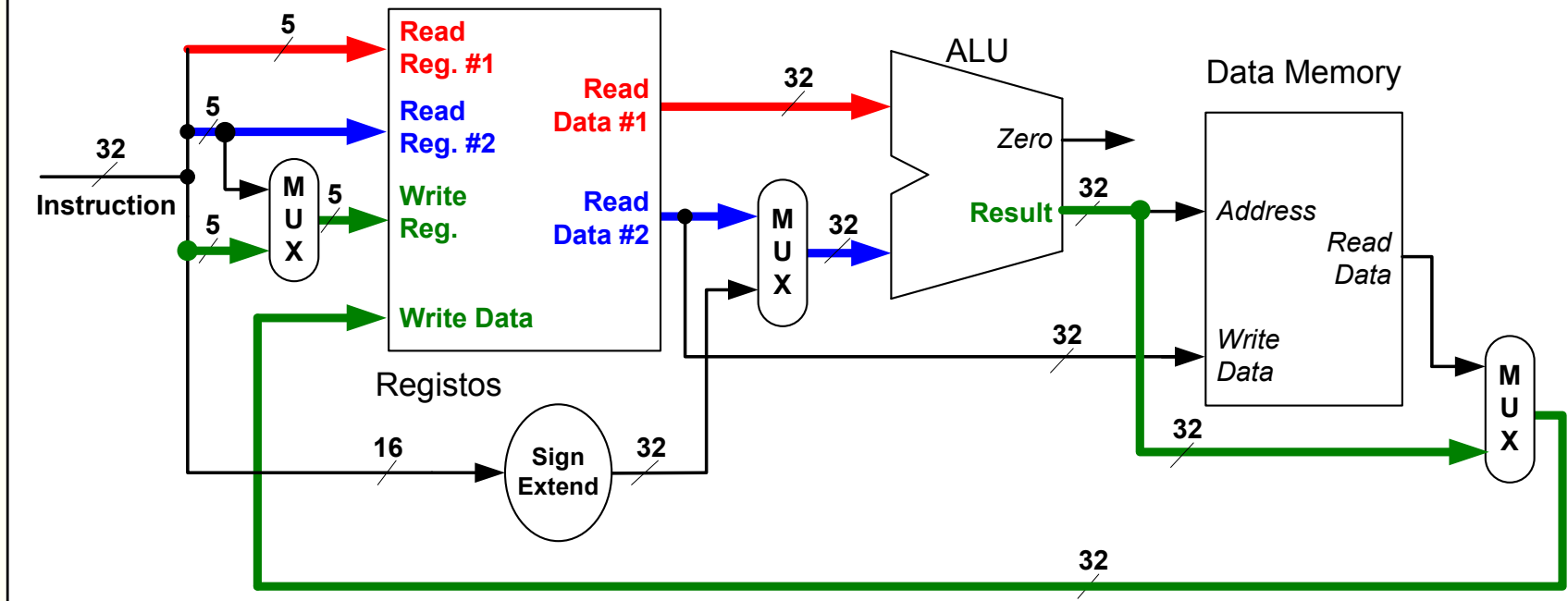
Multiplexers para adequar os recursos às necessidades (encaminhamento)



Implementação de um *Datapath* – juntando tudo

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

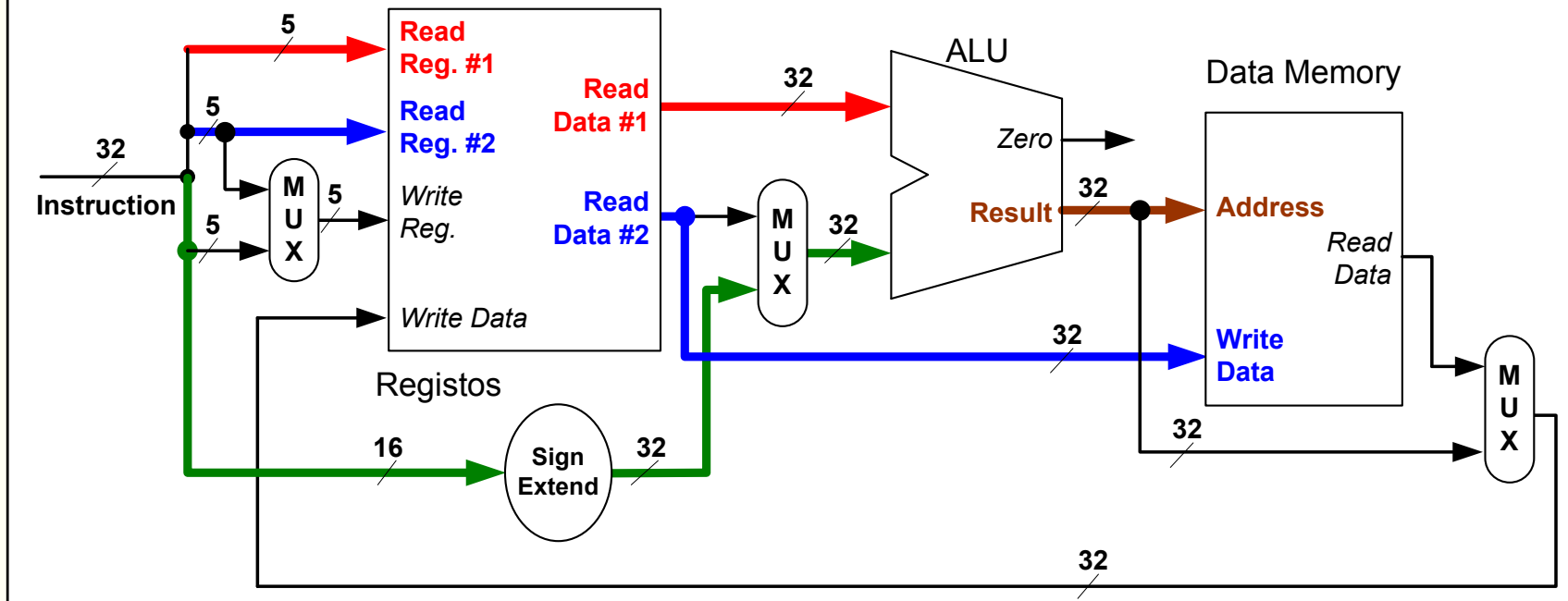
opcode (0)	rs (3)	rt (4)	rd (2)	shamt (0)	funct (32)
-----------------	-------------	-------------	-------------	----------------	-----------------



Implementação de um *Datapath* – juntando tudo

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

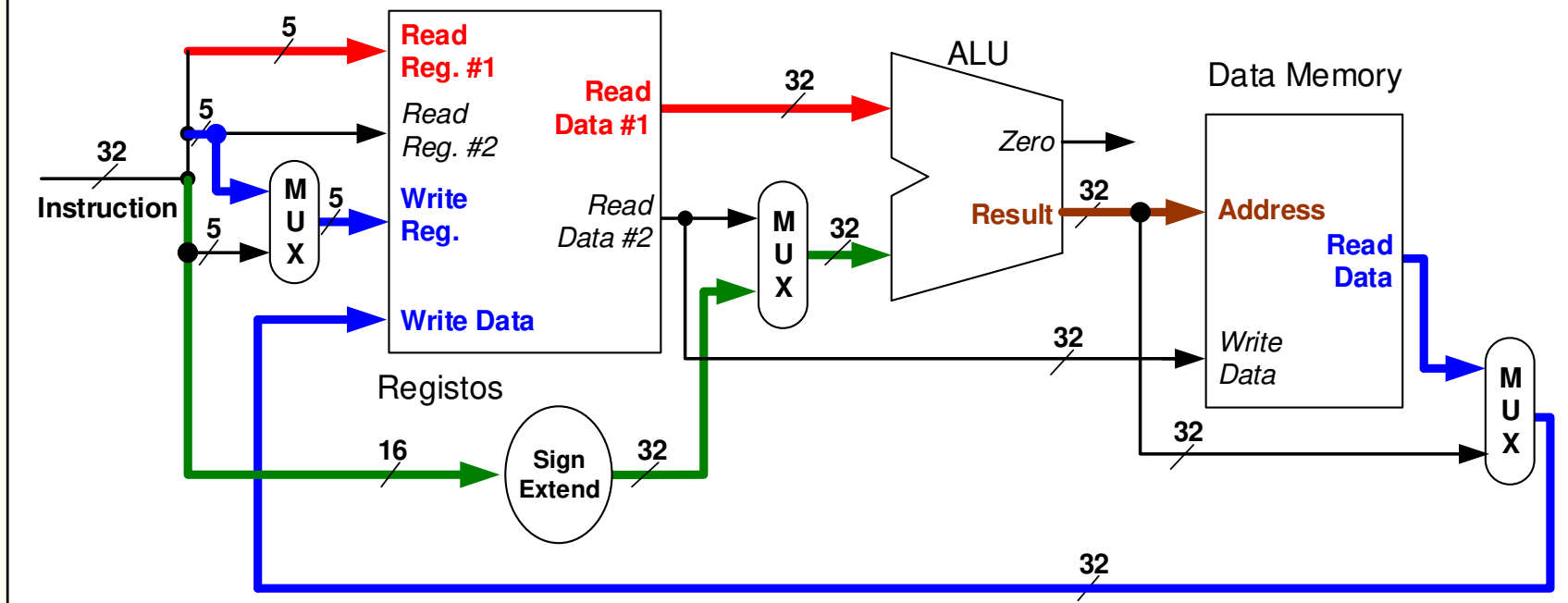
opcode (43)	rs (4)	rt (2)	offset (0x24)
------------------	--------------------	--------------------	---------------------------



Implementação de um *Datapath* – juntando tudo

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

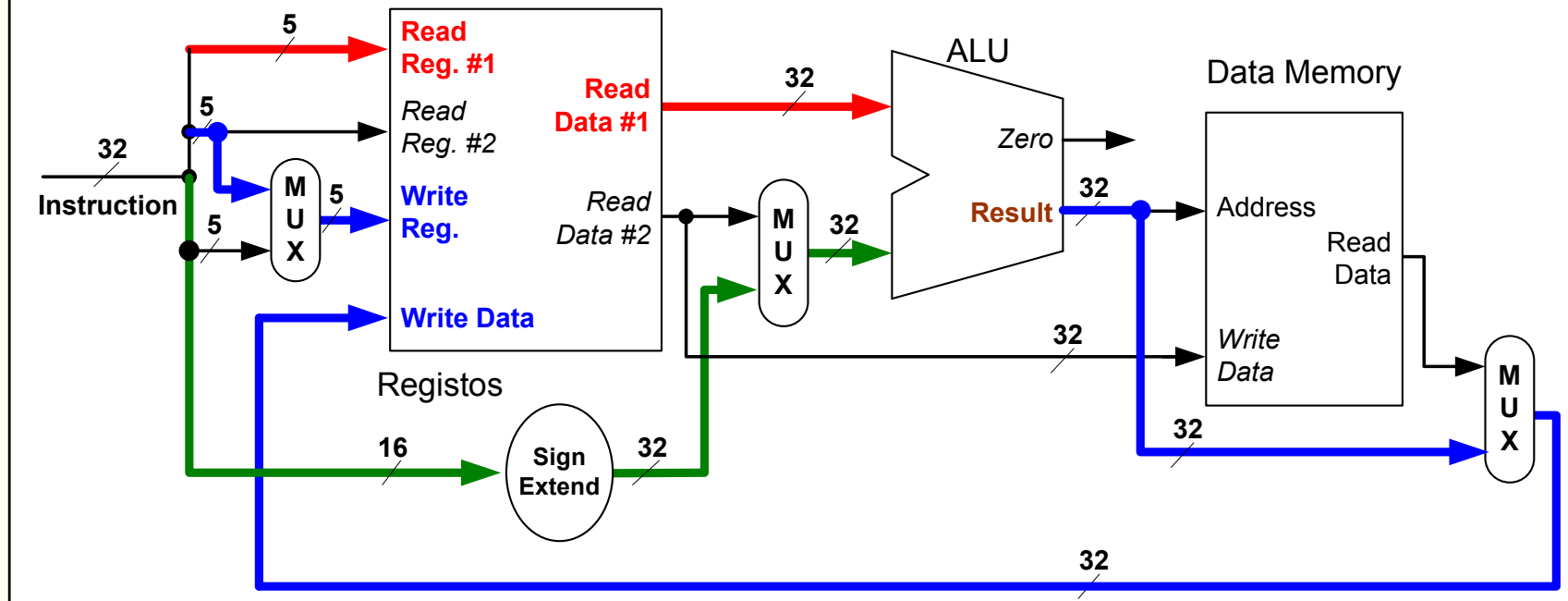
opcode (35)	rs (15)	rt (4)	offset (0x2F)
------------------	--------------	-------------	--------------------



Implementação de um *Datapath* – juntando tudo

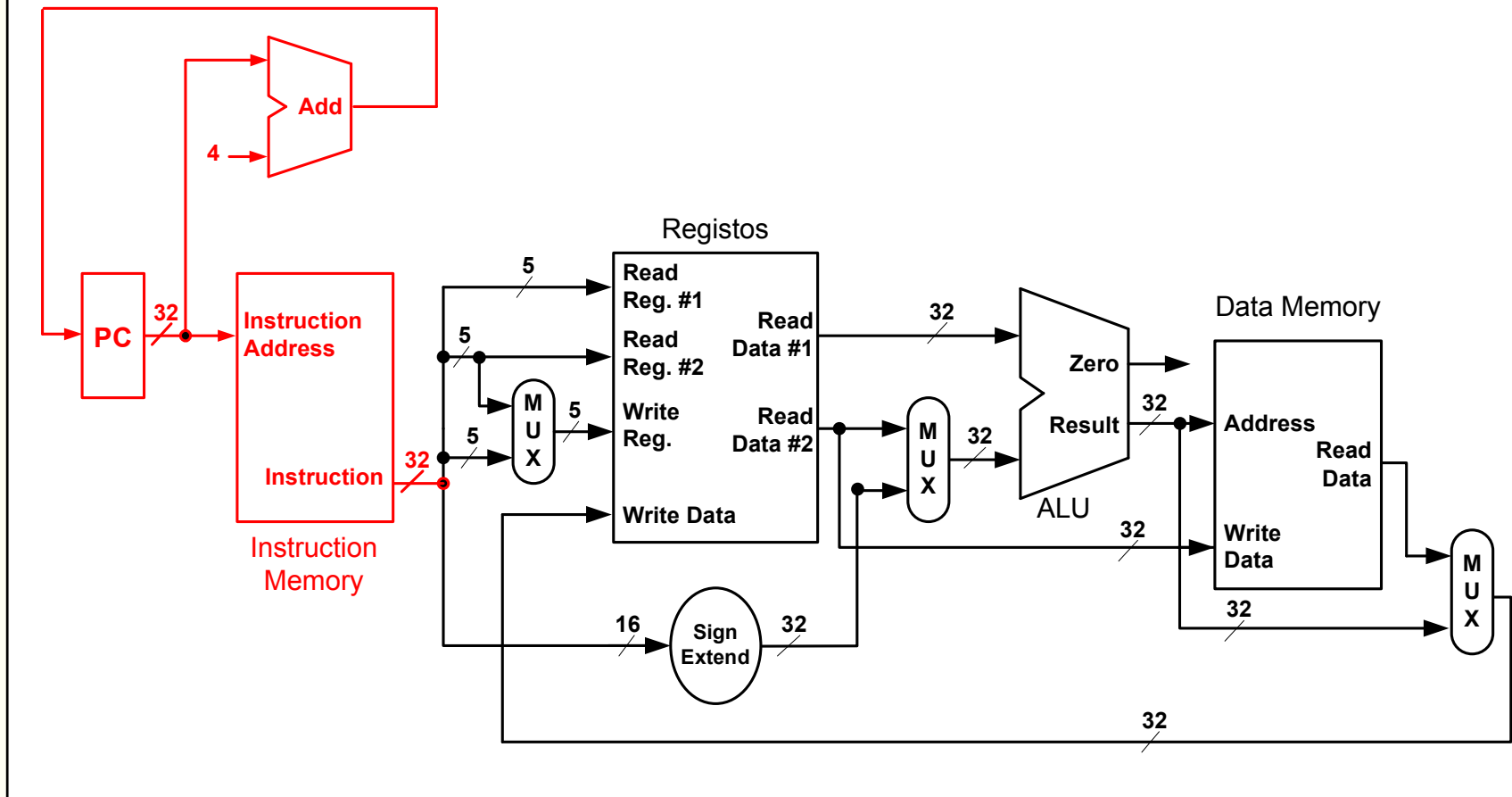
- Fluxo da informação na execução de uma instrução aritmética imediata. Exemplo: **addi \$4, \$15, 0x2F**

opcode (8)	rs (15)	rt (4)	offset (0x2F)
-----------------	--------------	-------------	--------------------



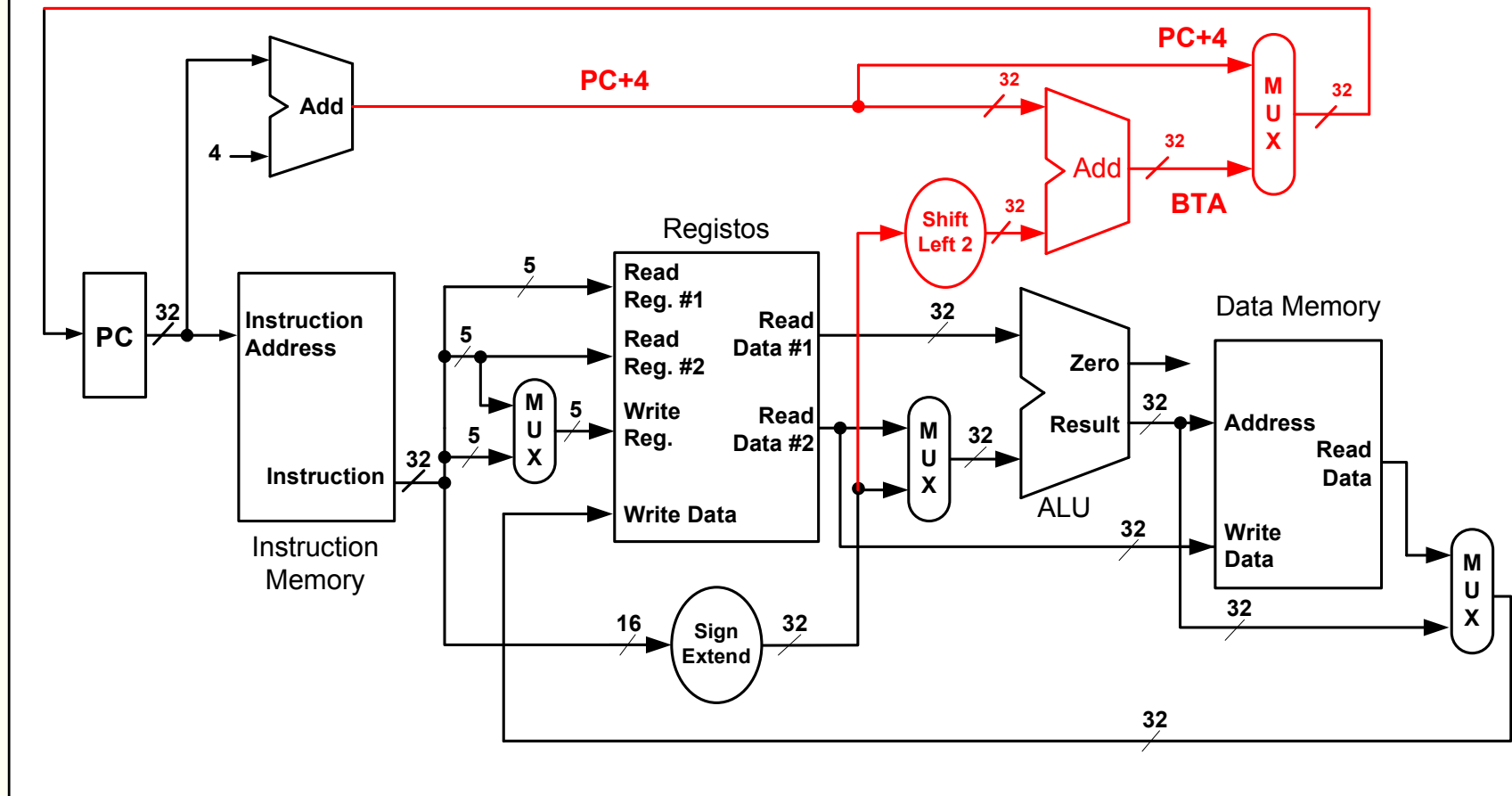
Implementação de um *Datapath* – juntando tudo

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



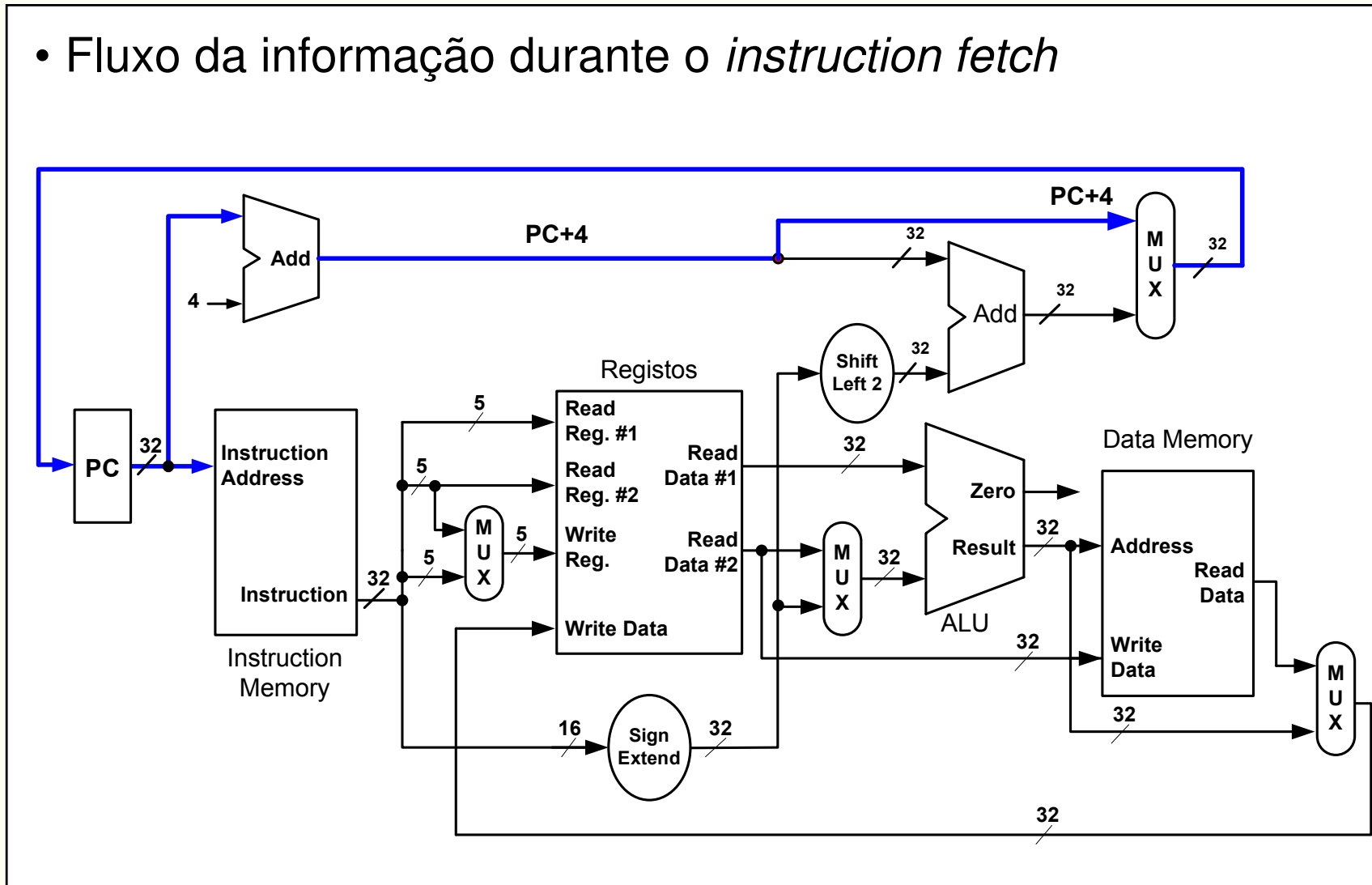
Implementação de um *Datapath* – juntando tudo

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



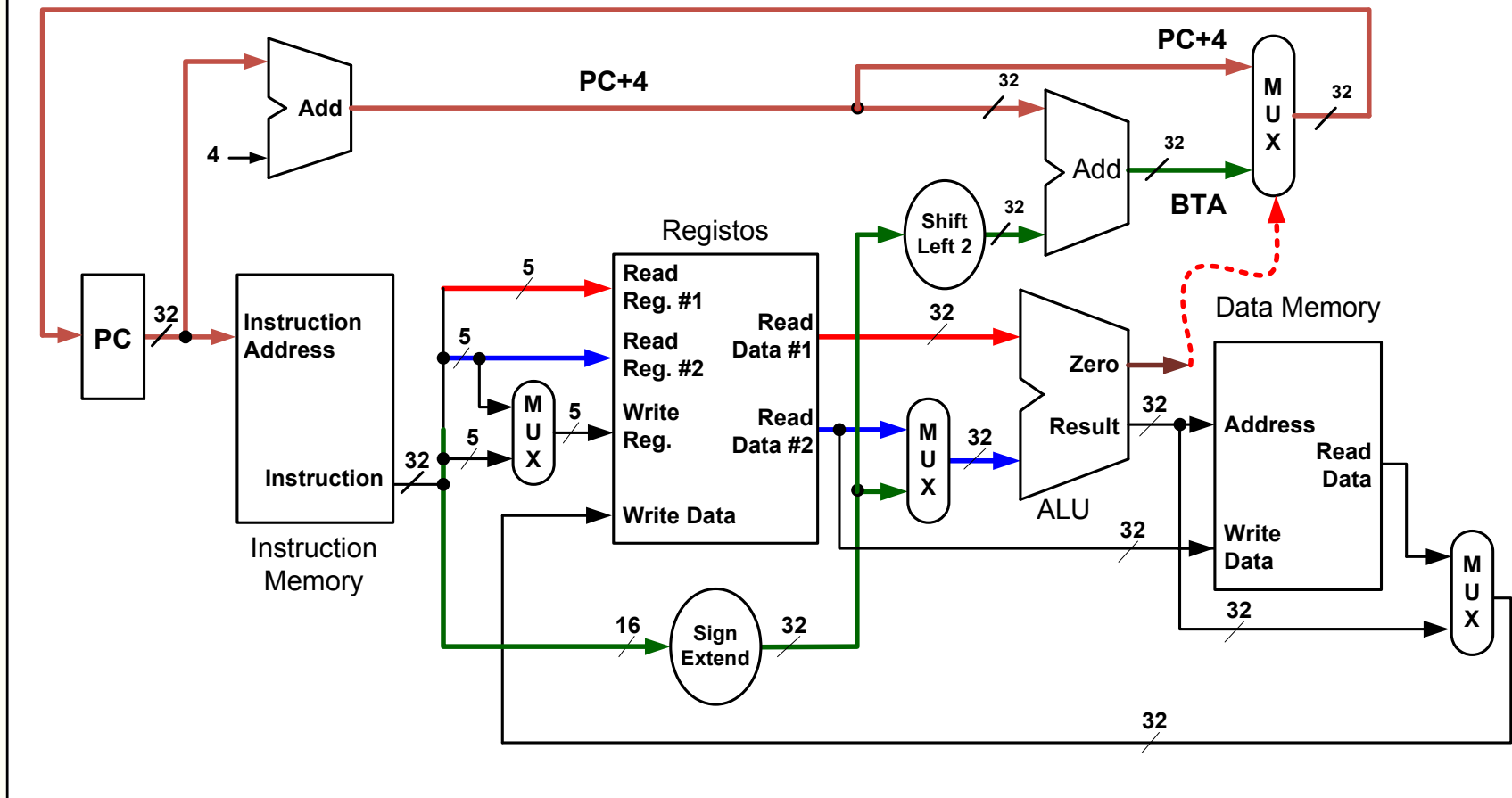
Implementação de um *Datapath* – juntando tudo

- Fluxo da informação durante o *instruction fetch*



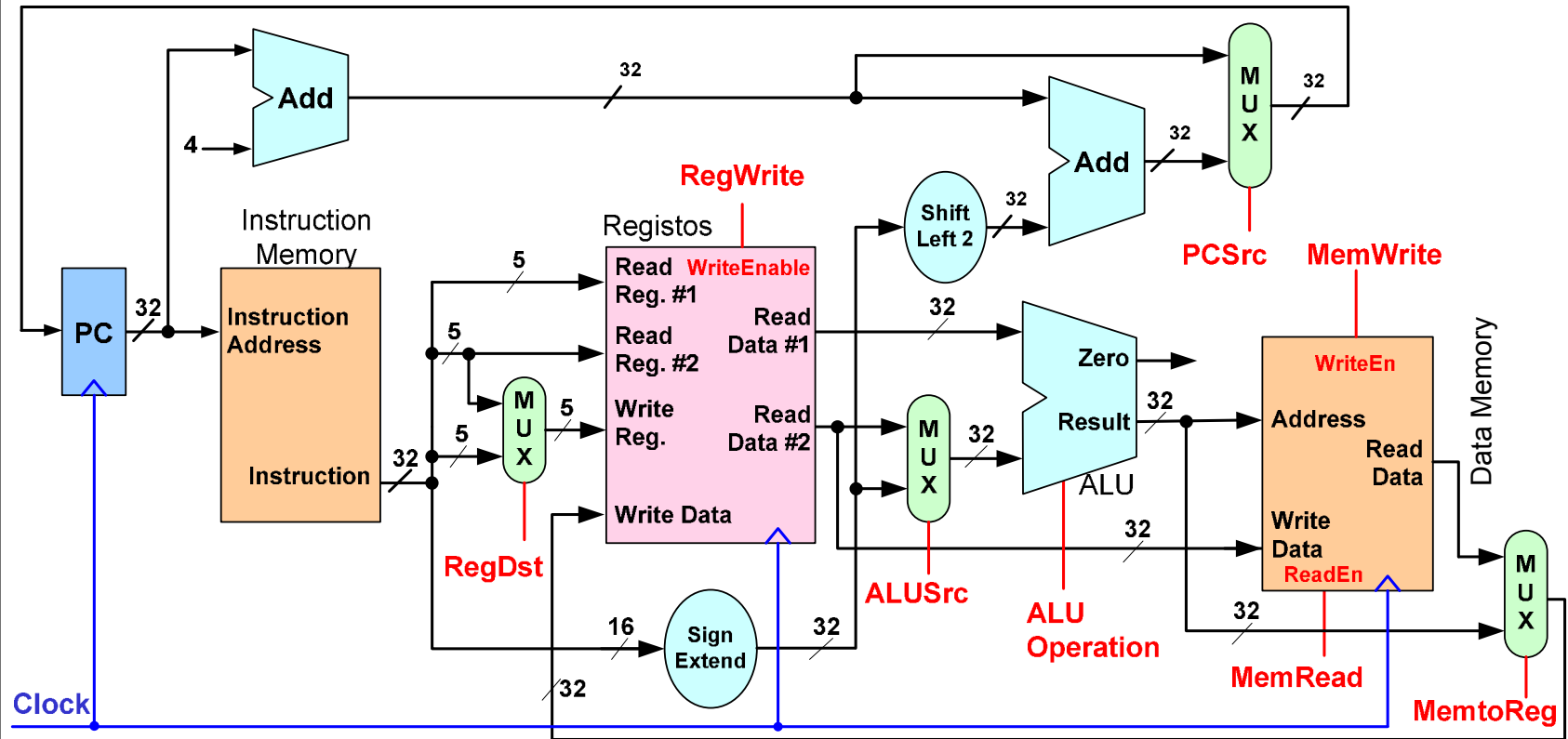
Implementação de um *Datapath* – juntando tudo

- Fluxo da informação na execução de uma instrução de *branch*



Datapath single-cycle completo

- Datapath completo com identificação dos sinais de controlo



A utilização de memórias distintas para dados e código, permite a execução de cada instrução num único ciclo de relógio

Que alterações é necessário fazer para incluir a execução do "bne" ?



Aulas 13, 14 e 15

- Instanciação dos vários módulos do datapath recorrendo à linguagem de programa hardware VHDL

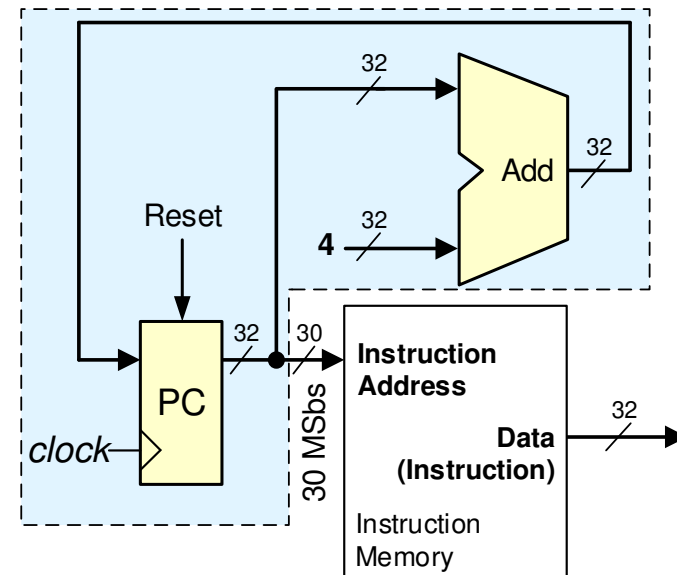
José Luís Azevedo, Bernardo Cunha, Arnaldo Oliveira, Pedro Lavrador



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;

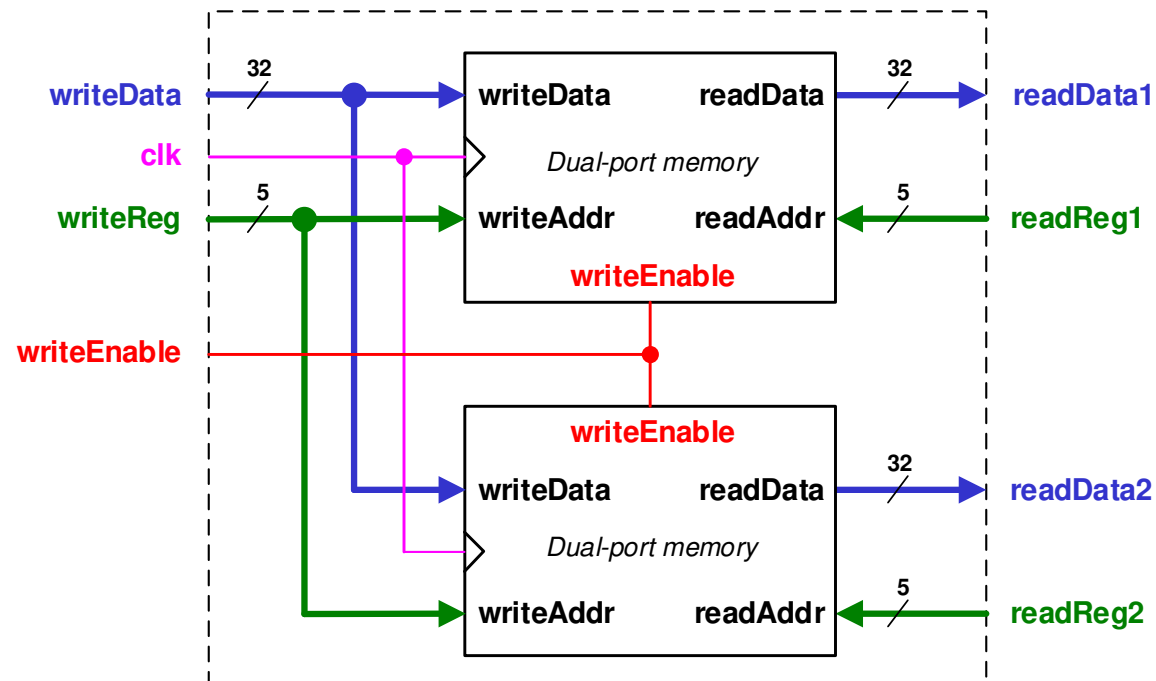
architecture Behavioral of instructionMemory is
    constant NUM_WORDS : positive := (2 ** ADDR_BUS_SIZE );
    subtype TData is std_logic_vector (31 downto 0);
    type TMemory is array (0 to NUM_WORDS - 1) of TData;
    constant s_memory : TMemory := (X"8C010000",  -- lw    $1,0($0)
                                     X"20210004",  -- addi  $1,$1,4
                                     X"AC010004",  -- sw    $1,4($0)
                                     others => X"00000000");

begin
    readData <= s_memory(to_integer(unsigned(address)));
end Behavioral;
```



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 registros (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 registros (dual-port memory) – VHDL

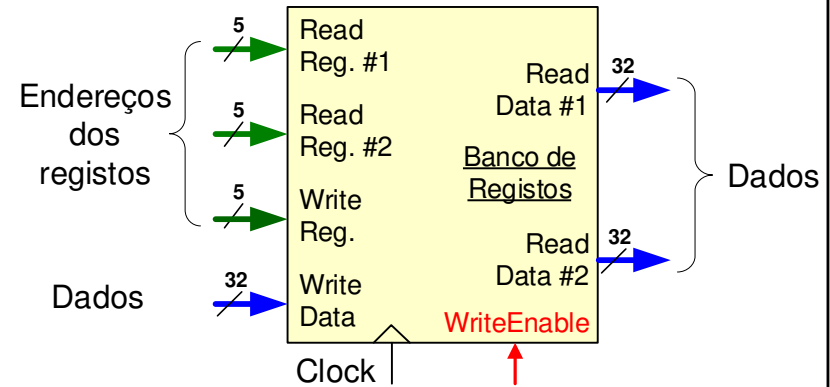
```
architecture Behavioral of DP_Memory is
    subtype TDataWord is std_logic_vector(WORD_BITS-1 downto 0);
    type TMem is array (0 to 2**ADDR_BITS-1) of TDataWord;
    signal s_memory : TMem := (others => (others => '0'));
begin
    process(clk, writeEnable) is
    begin
        if(rising_edge(clk) ) 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 registros – VHDL

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

```
entity RegFile is  
  port (clk          : in  std_logic;  
        writeEnable  : in  std_logic;  
        writeReg     : in  std_logic_vector( 4 downto 0 );  
        writeData    : in  std_logic_vector(31 downto 0 );  
        readReg1     : in  std_logic_vector( 4 downto 0 );  
        readReg2     : in  std_logic_vector( 4 downto 0 );  
        readData1    : out std_logic_vector(31 downto 0 );  
        readData2    : out std_logic_vector(31 downto 0 ));  
end RegFile;
```



Banco de registros – VHDL

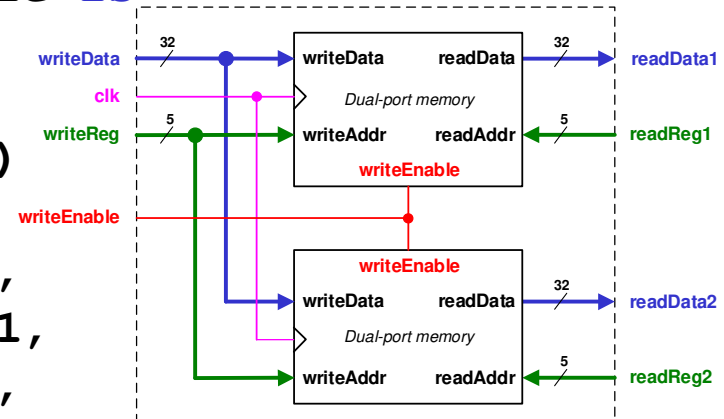
```
architecture Structural of RegFile is
begin
```

```
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;
```



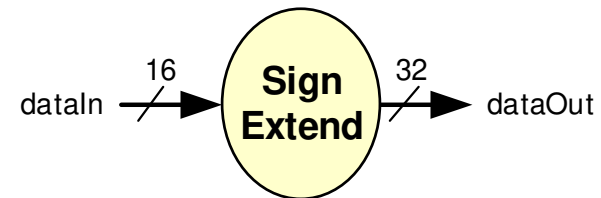
Módulo para separação dos campos da instrução – VHDL

```
entity InstrSplitter is
  port( instruction : in  std_logic_vector(31 downto 0);
        opcode      : out std_logic_vector(5  downto 0);
        rs          : out std_logic_vector(4  downto 0);
        rt          : out std_logic_vector(4  downto 0);
        rd          : out std_logic_vector(4  downto 0);
        shamt       : out std_logic_vector(4  downto 0);
        funct       : out std_logic_vector(5  downto 0);
        imm         : out std_logic_vector(15 downto 0);
        jAddr       : out std_logic_vector(25 downto 0));
end InstrSplitter;
architecture Behavioral of InstrSplitter is
begin
  opcode <= instruction(31 downto 26);
  rs     <= instruction(25 downto 21);
  rt     <= instruction(20 downto 16);
  rd     <= instruction(15 downto 11);
  shamt  <= instruction(10 downto  6);
  funct  <= instruction( 5 downto  0);
  imm    <= instruction(15 downto  0);
  jAddr  <= instruction(25 downto  0);
end Behavioral;
```



Módulo de extensão de sinal – VHDL

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



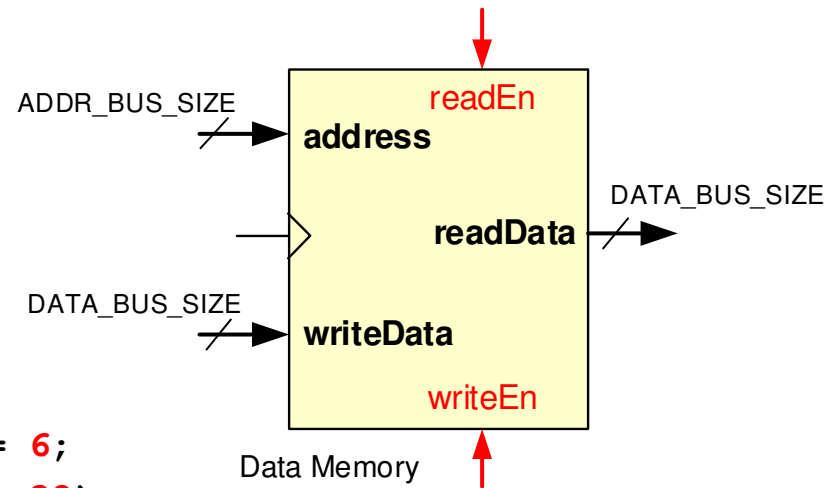
```
entity SignExtend is  
    port (dataIn  : in  std_logic_vector(15 downto 0);  
          dataOut : out std_logic_vector(31 downto 0));  
end SignExtend;
```

```
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 => '-');
end Behavioral;
```



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;
```

