UNIVERSIDADE FEDERAL DO VALE DO SÃO FRANCISCO

Especificação do Projeto de Processador RISC

myCPU

Neste documento é apresentada a especificação do projeto da disciplina Organização e Arquitetura de Computadores II. Tal documento tem como base a experiência adquirida durante o doutoramento do professor da disciplina.

1. INTRODUÇÃO

Esta especificação é composta por três sessões. Na primeira é apresentada a especificação e as várias etapas de apresentação do projeto. Na segunda é dada uma visão da CPU que será implementada. Nesta sessão também está presente o repertório de instruções que o processador desenvolvido deve conter e a descrição dos componentes fornecidos pelo professor da disciplina. Na última sessão é apresentada a especificação do relatório do projeto. Ainda está presente um anexo que ensina como configurar e carregar a memória que irá compor o projeto.

OBSERVAÇÂO

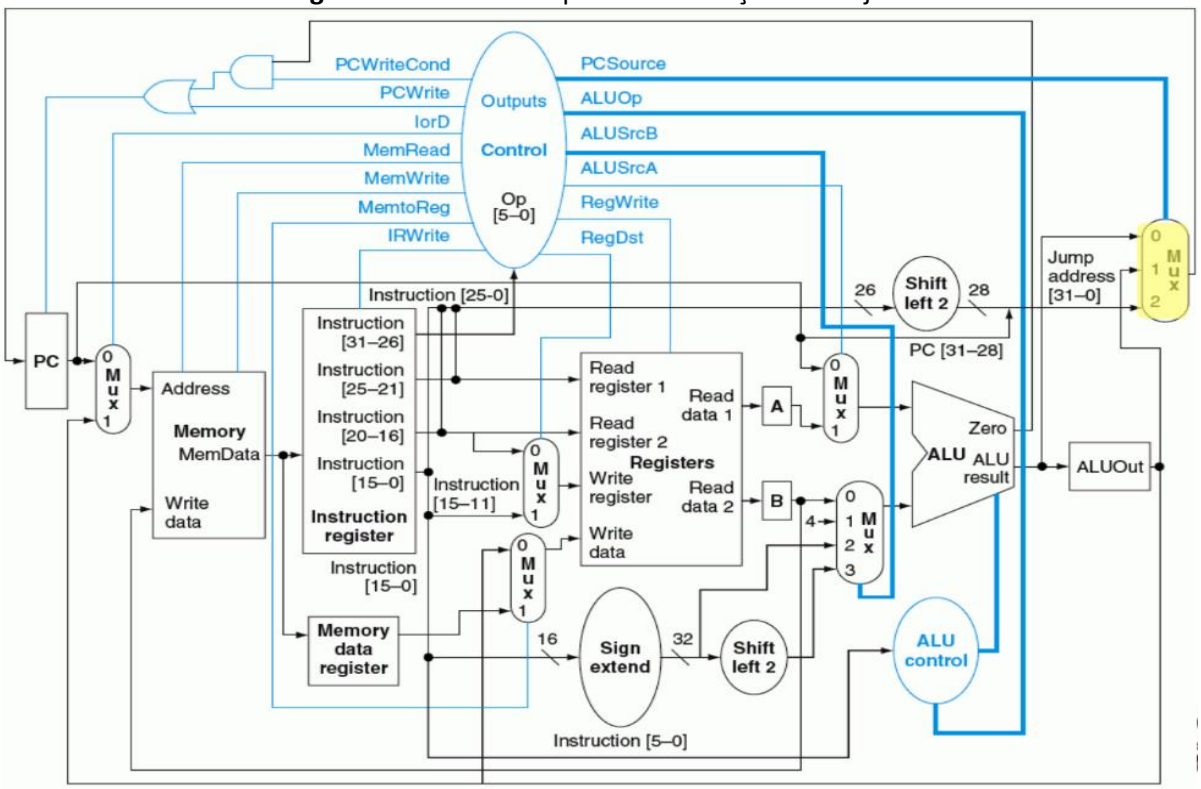
É dever de cada equipe ler e entender o que está descrito neste documento. Caso existam dúvidas, o professor deverá ser consultado para esclarecer qualquer dúvida que possa existir durante a execução do projeto. Ao final de cada sessão existem algumas observações que devem ser atentamente seguidas. Caso algumas dessas observações não sejam seguidas, a nota no projeto será coerente com a falha cometida.

Os alunos devem ter profissionalismo ao fazer o projeto. E importante lembrar que as regras servem para facilitar o trabalho dos alunos e do professor e, caso elas sejam cumpridas a risca, não haverá problemas com relação à correção.

2. ESPECIFICAÇÃO DA CPU

O projeto consiste no desenvolvimento de um processador, a qual poderá ser feita com base na descrita na Figura 1. O processador deverá incluir três blocos básicos: **Unidade de processamento**, **Unidade de Controle** e **Memória**. A seguir é dada uma descrição sucinta do processador a ser projetado.

Figura 1: Circuito Base para a Construção do Projeto



Inicialmente será projetado a **Unidade de Processamento** e a seguir a **Unidade de Controle**. A unidade de processamento será desenvolvida a partir da utilização dos seguintes componentes já descritos em VHDL e fornecidos através da página da disciplina:

- ALU
- Registrador de deslocamento
- Registrador de 32 bits
- Registrador de instruções
- Banco de registradores
- Memória

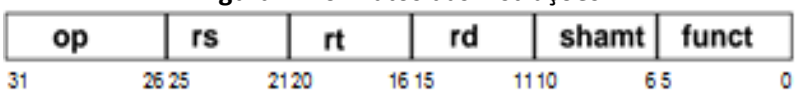
Após o projeto da unidade de processamento com a definição de todos os seus componentes e suas respectivas portas de entrada e saída, deverá ser projetada a unidade de controle, que será desenvolvida como uma máquina de estado finito.

O processador possui 32 registradores de propósito geral, cada um de 32 bits, como mostrado na Tabela 1. Um subconjunto das instruções do MIPS deverá ser desenvolvido, de acordo com esta especificação. As instruções estão descritas nesta sessão e agrupadas por formato. Um resumo dos formatos existentes pode ser visto na Figura 2.

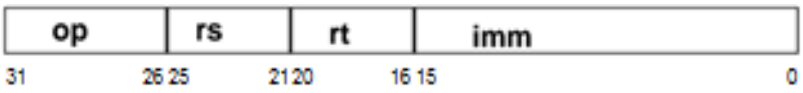
Tabela 1: Registradores

Registradores		
0	$zero	Valor constante igual a zero
1	$at	Assembler temporário.
2 - 3	$v0 - $v1	Retorno de função.
4 - 7	$a0 - $a3	Argumento de função.
8 - 15	$t0 - $t7	Variáveis temporárias.
16 - 23	$s0 - $s7	Variáveis salvas por uma função.
24 - 25	$t8 - $t9	Mais variáveis temporárias.
26 - 27	$k0 - $k1	Temporários para o sistema operacional.
28	$gp	Apontador global.
29	$sp	Apontador de pilha.
30	$fp	Apontador de quadro.
31	$ra	Endereço de retorno.

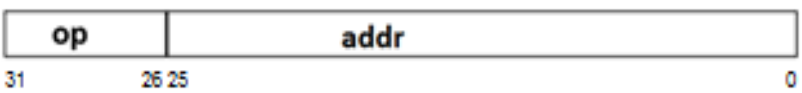
Figura 2: Formatos das Instruções



(a) Formato de instruções do tipo R



(b) Formato de instruções do tipo I



(c) Formato de instruções do tipo J

As tabelas a seguir apresentam todas as instruções que serão implementadas no projeto. Tabela 2 apresenta as instruções do tipo R

Tabela 2: Instruções do tipo R

Assembly	op	rs	rt	Rd	shamt	funct	Comportamento
add rd, rs, rt	0x0	rs	rt	Rd	0x0	0x20	rd <- rs + rt
and rd, rs, rt	0x0	rs	rt	Rd	0x0	0x24	rd <- rs & rt
jr rs	0x0	rs	-	-	0x0	0x8	PC <- rs
sll rd, rt, shamt	0x0	-	rt	Rd	shamt	0x0	rd <- rt << shamt
sllv rd, rt, rs	0x0	rs	rt	Rd	0x0	0x4	rd <- rt << rs
slt rd, rs, rt	0x0	rs	rt	Rd	0x0	0x2A	rd <- (rs < rt) ? 1 : 0
srl rd, rt, shamt	0x0	-	rt	Rd	Shamt	0x2	rd <- rt >> shamt
sub rd, rs, rt	0x0	rs	rt	Rd	0x0	0x22	rd <- rs – rt
or rd, rs, rt	0x0	rs	rt	Rd	0x0	0x25	rd <- rs | rt
xor rd, rs, rt	0x0	rs	rt	Rd	0x0	0x26	rd <- rs ^ rt

Tabela 3: Instruções do tipo I

Assembly	op	rs	rt	Offset/Immediate	Comportamento
addi rt, rs, imm	0x8	rs	rt	Imm	rt <- rs + imm
andi rt, rs, imm	0x9	rs	rt	Imm	rt <- rs + imm
beq rs, rt, offset	0x4	rs	rt	offset	Se rs = rt, PC += offset
bne rs, rt, offset	0x5	rs	rt	offset	if rs !=tt, PC += offset
lui rt, imm	0xF	-	rt	Imm	rt[31..15] <- imm; rt[15..0] <- 0
lw rt, offset(rs)	0x23	rs	rt	offset	rt <- mem [rs + offset]
slti rt, rs, imm	0xA	rs	rt	Imm	rt <- (rs < imm) ? 1 : 0
sw rt, offset(rs)	0x2B	rs	rt	offser	Mem[offset + rs] <- rt
ori rt, rs, imm	0xD	rs	rt	imm	rt <- rs | imm
xori rt, rs, imm	0xE	rs	rt	imm	rt <- rs ^ imm

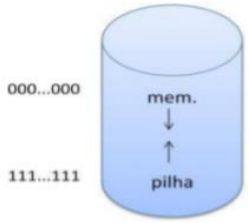
Tabela 4: Instruções do tipo J

Assembly	op	Offset	Comportamento
j offset	0x2	Offset	$ra = PC + 4, PC = offset
jal offset	0x3	Offset	rt <- rs + imm

2.1. Pilha

Para permitir a chamada de rotinas reentrantes e recursivas, será possível o armazenamento do registrador 31 ($ra) numa estrutura do tipo pilha a qual será implementada numa parte da memória como mostrado na Figura 3.

Figura 3: Estrutura da memória



O endereço do topo da pilha será guardado no registrador 29 ($sp) do banco de registradores, que funciona como o *Stack Point*. Quando o reset é ativado este registrador deverá apontar para o endereço onde começa a pilha. O salvamento e recuperação do registrador 31 ($ra) na pilha é sempre feito por software.

2.2. Desvios

Os desvios podem ser classificados como desvios condicionais ou incondicionais. Os desvios condicionais realizam algum teste ou verificação para, somente então executar o desvio. Os incondicionais sempre executam o desvio. O cálculo do endereço de destino é realizado diferentemente nos dois tipos de desvios:

2.2.1. Desvios condicionais

O valor de offset é multiplicado por 4 e somado ao PC atual. O resultado desta operação é o endereço de destino do salto (desvio), caso o resultado do teste condicional seja verdadeiro.

Exemplo: beq $t0, $t1, $t2 – Supondo que $t0 e $t1 sejam, ambos, iguais a 0x3A e que o PC atual seja 0x1C, a próxima instrução a ser executada é a que ocupa a posição (0xF * 4) + 0x1C, que é a 0x58.

Tabela 5: Desvios condicionais

Assembly	Descrição
beq rs, rt, offset	Realiza o desvio se rs for igual a rt
bne rs, rt, offset	Realiza o desvio se rs for diferente de rt

2.2.2. Desvios incondicionais

No caso de instruções que utilizem offset, este deve ser multiplicado por 4, resultando em um endereço de 28 bits. Como todo endereço deve possuir 32 bits, os 4 bits restantes vêm do PC atual, representando os 4 bits mais significativos do endereço de destino. No caso de instruções que utilizam registrador, o conteúdo do registrador, o conteúdo do registrador já é o próprio endereço de destino.

Exemplo 1: jal 0x1A2B – Supondo que o PC atual seja igual a 0xBA12FA00, o endereço de destino será igual a [(PC & 0xF0000000) | (0x1A2B * 0x4)], que é igual a 0Xb00068ac. Note que os 4 bits mais significativos do PC (1011) foram conservados.

Exemplo 2: jr $t0 – Supondo que o $t0 seja igual a 0x1A2B, o endereço de destino será igual ao próprio valor de $t0, neste caso, 0x1A2B.

Tabela 5: Desvios condicionais

Assembly	Descrição
j offset	Desvia, incondicionalmente, para o endereço calculado
jal offset	Desvia para endereço calculado, porém salva o endereço da instrução subsequente ao jal (endereço de retorno) no registrador 31.
jr rs	Desvio incondicional para o endereço apontado por rs.

2.3. Componentes Fornecidos

Nesta sessão são descritos os seis componentes do projeto que são fornecidos. Tais componentes poderão ser baixados a partir da página da disciplina.

2.3.1. Memória

A memória usada no projeto possuirá palavras de 32 bits com endereçamento por byte. Apesar de o endereço possuir 32 bits, a memória só possui 256 bytes. As entradas e saídas da memória são:

Figura 4: Entidade da Memória

```
entity ram is
  generic(
    DATA_WIDTH    : integer := 8;
    ADDRESS_WIDTH : integer := 8;
    DEPTH         : integer := 256
  );
  port(
    address    : in std_logic_vector(31 downto 0);
    clock      : in std_logic;
    we         : in std_logic;
    data_in    : in std_logic_vector(31 downto 0);
    data_out   : out std_logic_vector(31 downto 0)
);
end ram;
```

As portas de entrada ***address***, ***data_in***, ***clock***, ***we*** e ***data_out*** representam respectivamente o endereço de 32 bits de entrada da memória, a entrada de dados da memória (32 bits), o bit de clock comum a todo o processador, o bit que diz se vai ler ou escrever e a saída de dados da memória (32 bits).

Peculiaridades da Memória
1. Enquanto o bit *we* estiver com o valor 0 (zero) ele estará lendo, quando estiver com o valor 1 (um) ele estará escrevendo. 2. A memória está trigada na subida do *clock*. 3. Ao se fazer uma requisição de leitura, o valor pedido só estará disponível no segundo pulso de *clock* após o *clock* onde foi feito a requisição. 4. A escrita leva apenas um ciclo. 5. Instruções e dados serão armazenados na memória usando a estratégia *big-endian*.

2.3.2. Registrador de 32 bits

Para armazenar instruções e dados, bem como o endereço de instruções serão utilizados registradores de 32 bits, conforme ilustrado abaixo.

Observação
O dado que entra em ***d*** só passa para ***q*** quando o registrador passar 1 ciclo de *clock* com o sinal *load* ativo (*load* = 1)

Figura 5: Entidade do Registrador de 32 bits

```
entity reg32 is
  port(
    d       : in std_logic_vector(31 downto 0);
    clock   : in std_logic;
    reset   : in std_logic;
    load    : in std_logic;
    q       : out std_logic_vector(31 downto 0)
);
end reg32;
```

2.3.3. Banco de registradores

O banco de registradores é composto por 32 registradores de 32 bits. Dois registradores podem ser visíveis simultaneamente.

A leitura dos registradores é combinacional, isto é, se os valores nas entradas ***rr1*** ou ***rr2*** forem alteradas os valores nas saídas ***rd1*** ou ***rd2*** podem ser alterados. O registrador a ser escrito é selecionado pela entrada ***wr*** e quando a entrada ***rw*** é ativada (igual a 1) o registrador selecionado recebe o conteúdo da entrada wd. O sinal de reset limpa todos os registradores e é assíncrono.

Figura 6: Entidade do Banco de Registradores

```
entity registers is
  port(
    clock   : in std_logic;
    reset   : in std_logic;
    rr1     : in std_logic_vector(31 downto 0);  -- read register 1
    rr2     : in std_logic_vector(31 downto 0);  -- read register 2
    rw      : in std_logic;                      -- read or write
    wr      : in std_logic_vector(4 downto 0);   -- register for write
    wd      : in std_logic_vector(31 downto 0);  -- write data
    rd1     : out std_logic_vector(31 downto 0); -- read data 1
    rd2     : out std_logic_vector(31 downto 0); -- read data 2
);
end registers;
```

2.3.4. Registrador de deslocamento

O registrador de deslocamento deve ser capaz de deslocar um número inteiro de 32 bits para esquerda e para a direita. No deslocamento a direita o sinal pode ser preservado ou não.

O número de deslocamentos pode variar entre 0 e 32 e é especificado na entrada ***n*** (5 bits) do registrador de deslocamento. A funcionalidade desejada do registrador de deslocamento é especificada na entrada ***shift*** conforme Tabela 6. As atividades discriminadas na entrada ***shift*** são síncronas com o ***clock*** e o ***reset*** é assíncrono e ele funciona de forma semelhante como um registrador (demora 1 ciclo para sair o valor certo).

Figura 7: Entidade do Registrador de Deslocamento

```
entity shift_n is
  port(
    clock   : in  std_logic;
    reset   : in  std_logic;
    shift   : in  std_logic_vector (2 downto 0);
    n       : in  std_logic_vector (4 downto 0);
    d       : in  std_logic_vector (31 downto 0);
    q       : out std_logic_vector (31 downto 0)
  );
END shift_n;
```

Tabela 6: Funcionalidade do Registrador de Deslocamento

Shift	Descrição
000	Nada a fazer
001	Load no registrador
010	Shift à esquerda n vezes
011	Shift lógico à direita n vezes
100	Shift aritmético à direita n vezes
101	Rotação à direita n vezes
110	Rotação à esquerda n vezes

2.3.5. Unidade lógica e aritmética

A unidade lógica e aritmética (ALU) é um circuito combinacional que permite a operação com números de 32 bits na notação complemento de dois. A funcionalidade é especificada pela entrada ***f*** conforme descrito na Tabela 7.

Figura 8: Entidade da ALU

```
entity alu32 is
  port (
    a           : in  std_logic_vector (31 downto 0);
    b           : in  std_logic_vector (31 downto 0);
    f           : in  std_logic_vector (2 downto 0);
    s           : out std_logic_vector (31 downto 0);
    o           : out std_logic;
    n           : out std_logic;
    z           : out std_logic;
    eq          : out std_logic;
    gt          : out std_logic;
    lt          : out std_logic
  );
end alu32;
```

Tabela 7: Funcionalidade da ALU

Função	Operação	Descrição	Flags
000	S = A	Carrega A	Z, N
001	S = A + B	Soma	Z, N, O
010	S = A – B	Subtração	Z, N, O
011	S = A and B	And lógico	Z
100	S = A + 1	Incremento de A	Z, N, O
101	S = not A	Negação de A	Z
110	S = A xor B	Ou exclusivo	Z
111	S = A comp B	Comparação	EQ, GT, LT

3. METODOLOGIA DE PROJETO E CRONOGRAMA DE AVALIAÇÃO

Nesta sessão são apresentados os procedimentos que devem ser seguidos para o desenvolvimento das várias etapas do projeto do processador descrito na sessão anterior desta especificação. Para cada uma das etapas os grupos devem apresentar a implementação em VHDL do projeto da unidade de processamento e o projeto da unidade de controle.

3.1. Primeira Etapa: Projetando a Busca da Instrução e Execução de algumas instruções

Nesta etapa os grupos apresentarão a implementação da etapa de busca da instrução na memória e a execução de algumas instruções. Os alunos deverão apresentar a unidade de processamento que faz a busca e a unidade de controle. A implementação deverá ser simulada de forma a ser possível visualizar o valor do PC, a informação lida da memória, o registrador de instruções e os estados da máquina de estado e as saídas que explicitam essas informações devem possuir os nomes especificados a seguir.

Esta etapa corresponde a metade (50%) da nota referente ao projeto. Caso o grupo não consiga implementar a busca, a nota desta etapa será zerada e será fornecido para o grupo um projeto com a busca já implementada, mas ainda será necessário que o grupo implemente todo o repertório de instruções do projeto.

Os alunos deverão apresentar a unidade de processamento e a unidade de controle que realozam a busca de instruções e a execução das instruções descritas nas Tabelas 8, 9 e 10. A implementação em VHDL também deverá ser apresentada e deverá ser simulada de forma a ser possível visualizar o valor do PC, a informação lida e escrita da memória, o registrador de instruções, o conteúdo dos registradores lidos, o conteúdo a ser escrito no registrado e o conteúdo do registrado que se localiza na saída da ALU.

Tabela 8: Instruções do tipo R para Primeira Etapa

Assembly	op	rs	Rt	Rd	shamt	funct	Comportamento
add rd, rs, rt	0x0	rs	Rt	Rd	0x0	0x20	rd <- rs + rt
and rd, rs, rt	0x0	rs	Rt	Rd	0x0	0x24	rd <- rs & rt
sub rd, rs, rt	0x0	rs	Rt	Rd	0x0	0x22	rd <- rs – rt
xor rd, rs, rt	0x0	rs	Rt	Rd	0x0	0x26	rd <- rs ^ rt

Tabela 9: Instruções do tipo I para Primeira Etapa

Assembly	op	rs	rt	Offset/Immediate	Comportamento
beq rs, rt, offset	0x4	rs	rt	offset	Se rs = rt, PC += offset
bne rs, rt, offset	0x5	rs	rt	offset	if rs !=tt, PC += offset
lui rt, imm	0xF	-	rt	Imm	rt[31..15] <- imm; rt[15..0] <- 0
lw rt, offset(rs)	0x23	rs	rt	offset	rt <- mem [rs + offset]
sw rt, offset(rs)	0x2B	rs	rt	offser	Mem[offset + rs] <- rt

Tabela 10: Instruções do tipo J para Primeira Etapa

Assembly	op	Offset	Comportamento
j offset	0x2	Offset	$ra = PC + 4, PC = offset

Observação
É obrigatória a presença das seguintes saídas na simulação no *waveform* exatamente com os mesmo nomes citados na Tabela 11, sob a possibilidade de haver penalidades caso não haja pinos ou estejam com os nomes diferentes. Podem haver outras saídas, desde que estas estejam presentes.

Sinais	**Descrição**
signal_memData	Mostra o valor na saída da memória
signal_address	Mostra o endereço onde será feita a leitura ou escrita na memória. O sinal deve estar após o mux.
signal_writeDataMem	Valor que vai ser escrito na memória
signal_writeRegister	Mostra o dado que vai ser escrito no registrador
signal_MDR	Mostra o dado presente no Memory Data Register
Signal_alu	Mostra o valor que está saindo da ALU
Signal_aluOut	Mostra o valor presente na saída do Registrador AluOut
signal_PC	Valor presente no PC
signal_RegDesloc	Valor presente no registrador de deslocamento
signal_wr	Sinal que controla se o dado será lido ou escrito na memória
signal_RegWrite	Sinal que controla a escrita/leitura no banco de registradores
Signal_IRWrite	Sinal que controla a escrita no registrador de instruções

3.2.